Practical Architectures for Survivable Systems and Networks: Phase-One Final Report

Approved for public release

©Copyright 1999 SRI International, freely available for noncommercial reuse

January 28, 1999 including minor corrections

Peter G. Neumann
Computer Science Laboratory
SRI International, Room EL-243
333 Ravenswood Avenue
Menlo Park CA 94025-3493
Telephone: 1-650-859-2375
Fax: 1-650-859-2844
E-mail: `Neumann@CSL.sri.com`
`http://www.csl.sri.com/~neumann/`

Acknowledgment of Support and Disclaimer: This report is based upon work supported by the US Army Research Laboratory, under Contract DAKF11-97-C-0020. Any opinions, findings, conclusions, or recommendations expressed herein are those of the author and do not necessarily reflect the views of the U.S. Army Research Laboratory. Government contacts are LTC Paul Walczak (`PWalczak@arl.army.mil`), 1-301-394-3862, and Anthony Barnes (`TBarnes@arl.army.mil`), 1-732-427-5099.

NOTE: This document represents a somewhat personal view of some potentially effective approaches toward developing, configuring, and operating highly survivable system and network environments. It will continue to evolve throughout 1999 during the second phase of the work. Copies are available on-line at

`http://www.csl.sri.com/~neumann/arl-one.html`
`http://www.csl.sri.com/~neumann/arl-one.ps`
`http://www.csl.sri.com/~neumann/arl-one.pdf`

Constructive feedback is always welcome. Many thanks.

19991008 033

DTIC QUALITY INSPECTED 4

Preface

Abstract: This report summarizes the analysis of survivability-related requirements and their interdependence. It also identifies inadequacies in existing commercial systems and the absence of components that hinder the attainment of survivability. It recommends specific architectural structures and other approaches that can help overcome those inadequacies.

The field of endeavor addressed in this report is inherently open ended. New research results and new software components are emerging at a rapid pace. For this reason, the report stresses fundamentals, and is intended to be a guide to certain principles and architectural directions whose systematic use can lead to better survivability. In that spirit, the report is intended to serve as a coherent resource from which many further resources can be gleaned by following the cited references and URLs.

The report is quite modest in its intent. It does not try to solve all the problems of how to develop, maintain, and use highly survivable systems and networks. Those problems still must require many years of future research and development, but the report is a useful starting point.

This document will be updated periodically during the follow-on phase-two work in 1999, adding new material and amplifying the existing material with greater detail and new developments. The resulting document should be useful to developers of systems with critical requirements. It should also be useful in connection with anyone wanting to teach or learn the basics of system and network survivability. The Army Research Laboratory has sponsored an Information Survivability (InfoSurv) workshop in which some of this material was presented, and intends to do so again in the future. Several universities (Maryland, Penn, and possibly Georgia Tech) are already contemplating new courses on information survivability for as early as for Fall of 1999, and planning to integrate some of this material into mainstream curricula — at least in part inspired by this document. The appendix to this report characterizes some of the issues relating to curricula that include survivability. At this juncture, we have intentionally not tried to spell out specific course materials lecture by lecture, but rather have tried to provide basic directions that such courses might address.

BLANK PAGE

Contents

Preface i

Executive Summary ix

1 Introduction 1
- 1.1 Project Goals . . . 1
- 1.2 Fundamental Concepts . . . 2
  - 1.2.1 Survivability . . . 2
  - 1.2.2 Attributes of System Survivability . . . 3
  - 1.2.3 Trustworthiness, Dependability, and Assurance . . . 5
  - 1.2.4 Generalized Composition . . . 8
  - 1.2.5 Generalized Dependence . . . 9
  - 1.2.6 Generalized Survivability . . . 12
  - 1.2.7 Mandatory Policies for Security, Integrity, and Availability . . . 13
  - 1.2.8 Multilevel Survivability . . . 13
- 1.3 Compromisibility and Noncompromisibility . . . 14
- 1.4 Defenses Against Compromises . . . 19
- 1.5 Sources of Risks . . . 19
- 1.6 Some Relevant Case Histories . . . 21
- 1.7 Causes and Effects . . . 26

2 Threats to Survivability 30
- 2.1 Threats to Security . . . 30
  - 2.1.1 Bypasses . . . 31
  - 2.1.2 Pest Programs . . . 36
  - 2.1.3 Resource Misuse . . . 38
  - 2.1.4 Comparison of Attack Modes . . . 38
  - 2.1.5 Other Attack Methods . . . 40
  - 2.1.6 Personal-Computer Viruses . . . 41
- 2.2 Threats to Reliability . . . 42
- 2.3 Threats to Performance . . . 42
- 2.4 Perspective on Threats to Survivability . . . 42

3 Requirements and Their Interdependence 46
3.1 Survivability and its Subrequirements 47
3.1.1 Generalized Survivability 49
3.1.2 Security 50
3.1.3 Reliability and Fault Tolerance 52
3.1.4 Performance 53
3.2 System Requirements for Survivability 54
3.3 A System View of Survivability 57
3.4 Mapping Mission Requirements into Specifics 59

4 Systemic Inadequacies 60
4.1 System and Networking Deficiencies 60
4.2 Deficiencies in the Information Infrastructure 64
4.3 Other Deficiencies 64

5 Approaches for Overcoming Deficiencies 67
5.1 System and Networking Architectures 67
5.2 System and Networking Components 68
5.3 Configuration Management 69
5.4 Information Infrastructure 69
5.5 System Development Practice 70
5.6 Software Engineering Practice 70
5.7 Subsystem Composability 71
5.8 Formal Methods 72
5.9 Open-Source and "Free" Software 75
5.10 Integrative Paradigms 79
5.11 Fault Tolerance 79
5.12 Static System Analysis 80
5.13 Real-time Analysis of Behavior and Response 80
5.14 Standards 81
5.15 Research and Development 81
5.16 Education and Training 83
5.17 Organizations 83

6 Evaluation Criteria 85

7 DoD Attempts at Standardization 88
7.1 The Joint Technical Architecture 88
7.1.1 Goals of JTA Version 5.0 88
7.1.2 Analysis of JTA Version 5.0 89
7.1.3 JTA5.0 Section 6, Information Security 91
7.1.4 Augmenting the Army Architecture Concept 93
7.2 The DoD Goal Security Architecture 93
7.3 Joint Airborne SIGINT Architecture 94
7.4 An Open-Systems Process for DoD 95

8 Architectures for Survivability **96**
- 8.1 Structural Organizing Principles . . . 97
- 8.2 Architectural Structures . . . 101
- 8.3 Architectural Components . . . 104
  - 8.3.1 Secure Operating Systems . . . 104
  - 8.3.2 Networking Software . . . 105
  - 8.3.3 Encryption and Key Management . . . 105
  - 8.3.4 Authentication Subsystems . . . 106
  - 8.3.5 Trusted Paths and Resource Integrity . . . 106
  - 8.3.6 File Servers . . . 107
  - 8.3.7 Network Servers . . . 107
  - 8.3.8 Name Servers . . . 107
  - 8.3.9 Monitoring . . . 107
  - 8.3.10 Architectural Challenges . . . 108
- 8.4 The Mobile-Code Architectural Paradigm . . . 108
  - 8.4.1 Confined Execution Environments . . . 111
  - 8.4.2 Revocation and Object Currency . . . 113
  - 8.4.3 Proof-Carrying Code . . . 113
  - 8.4.4 Architectures Accommodating Untrustworthy Mobile Code . . . 113
- 8.5 Structural Architectures . . . 114
  - 8.5.1 Conventional Architectures . . . 114
  - 8.5.2 Multilevel Survivability with Minimized Trust . . . 114
  - 8.5.3 End-User System Components . . . 116

9 Implementing and Configuring for Survivability **119**
- 9.1 Developing Survivable Systems . . . 119
- 9.2 A Baseline Survivable Architecture . . . 120

10 Conclusions **122**
- 10.1 Recommendations for the Future . . . 123
- 10.2 Research and Development Directions . . . 124
- 10.3 Lessons Learned from Past Experience . . . 125
- 10.4 Architectural Directions . . . 128
- 10.5 Residual Vulnerabilities and Risks . . . 128
- 10.6 Applicability to the PCCIP Recommendations . . . 128
- 10.7 Future Work . . . 129
- 10.8 Final Comments . . . 130

Acknowledgments **131**

Appendix: Curriculum for Survivability **132**

A.1 Survivability-Relevant Courses **132**

A.2 Applicability of Remote Learning **135**

A.3 Summary of Education and Training Needs **137**

A.4 The Purpose of Education **137**

Noteworthy References **139**

Bibliography **140**

Index **169**

List of Tables

1.1 Illustrative Compromises 18

2.1 Types of Computer Misuse 33
2.2 Illustrative Reliability Threats 43

3.1 Some Survivability Attributes at Different Logical Layers 58

5.1 Defensive Measures 68

8.1 Typical Architectural Limitations 109
8.2 Architectural Needs 115

10.1 Survivability Curricula 133

List of Figures

2.1 Classes of Computer Misuse Techniques 32
3.1 Illustrative Subset of Requirements Hierarchy 48

Executive Summary

The Problem

Systems and networks with critical survivability requirements are extremely difficult to specify, develop, procure, operate, and maintain. They tend to be subject to many threats, laden with risks, and difficult to use wisely. We begin with several observations.

- The U.S. Government and the defense establishment have become increasingly dependent on commercially available systems — with all their warts, blemishes, and fundamental shortcomings. Unfortunately, these systems are typically not robust enough and not advancing fast enough for critical applications. Fortunately, a viable alternative for obtaining critical systems is emerging: the possibility of a disciplined effort to modify certain open-source software components and make them substantially more robust — particularly where commercial off-the-shelf products are not adequate.

- Commercially available systems are very poor with respect to security and reliability. They are even worse with respect to overall system and network survivability, which depends critically on security and reliability. Software components are often incompatible with one another, even when obtained from the same developer. Interoperability and reusability are much less than what should reasonably be expected. Compatibility with legacy systems is driving many systems into their lowest common denominators. Proprietary systems and proprietary interface standards generally make integration with other system types very difficult. They also make open analysis of those systems very difficult (promoting security by obscurity rather than security by in-depth analysis). Long-term compatible evolvability is a serious problem.

- System development practice is in general abysmal. (Recent examples of total fiascos in the development of U.S. Government systems include the cancellations of the FAA Air Route Traffic Control System development, the IRS Tax Systems Modernization effort, and the FBI NCIC-2000 fingerprint system development, representing the waste of billions of dollars.) A representative example of the very bad state of practice and the great difficulties inherent in trying to advance the state of commercial systems is given by the mere existence and pervasiveness of the Year-2000 problem, with the resulting enormous costs to attempt to fix billions of lines of code and the lingering doubts as to whether those attempts will be successful. The Y2K problem is just one more example of short-sighted system development practice, rather than a unique problem unto itself.

- The range of threats to survivability that must be considered is enormous, including hardware malfunctions, software flaws, environmental hazards, and malicious and accidental human acts. The level of awareness of threats, vulnerabilities, and risks is generally deplorable. Offensive information warfare is becoming recognized as a serious potential threat to survivability, but attempts to develop corresponding defenses are lagging badly. Many organizations (perhaps most visibly, the Pentagon) have been deluged with attacks on their Web sites and computer-communication infrastructures.

- The recent report of the President's Commission on Critical Infrastructure Protection (PCCIP) touches on the tip of an enormous iceberg. It observes that the survivability and integrity of the critical national infrastructures (such as telecommunications, power, energy, transportation, and finances) are very much at risk, and furthermore that these national infrastructures are highly interdependent. Whereas the report recognized that the critical national infrastructures depend critically on computers and communications, their recommendations touch only lightly on what might be done to strengthen the underlying computer-communication infrastructures. (Although the Commission's initial focus in telecommunications was narrowly aimed at the public switched network, the Commission's members did ultimately attempt to broaden their concerns to include the Internet and computer networking.) The PCCIP recommendations are important and must be considered very carefully, although there is a tendency toward increasing bureaucracy. (See Section 5.17 for more on the Commission and its follow-ons.)

- Although significant research and prototype-development efforts could help minimize many of the existing problems, and many new problems that need to be addressed, valuable R&D advances related to security, reliability, and survivability have in general been exceedingly slow in finding their way into practice. That must change.

Goals

These observations motivate a simple statement of the goals of our project:

- To surmount these realities, we seek to (1) make the requirements for survivability and its necessary subtended properties such as security and reliability explicit, (2) identify functionality whose absence currently prevents adequate satisfaction of those requirements, (3) explore techniques for designing and developing highly survivable systems and networks, despite the presence of untrustworthy subsystems and untrustworthy people — where untrustworthiness may encompass the lack of reliability, integrity, and correctness of behavior on the part of systems and people, and (4) recommend specific architectural structures and structural architectures that can lead to survivable systems and networks capable of either preventing or tolerating a wide range of threats.

Approach of the Report

It is absolutely essential to realize that there are no easy answers to achieving survivable systems and networks. This report does not pretend to be a cookbook. Cookbook approaches are doomed to fail, because of the intrinsic multidimensionality of the problem, the inadequacies of the existing infrastructures, the fact that the underpinnings are continually in flux, and the fact that no one solution or small set of solutions fits all applications. We cannot merely follow tried-and-true recipes, because no foolproof recipes exist. For these reasons, we emphasize the need for depth of understanding of the basic issues, the recognition and pervasive adherence to sensible principles, the fundamental importance of insights gleaned from past experience, and the urgency of pursuing significant R&D approaches and incorporating them into practical systems. Thus, we include many references to primary literature sources, with the hopes that diligent readers will pursue them. The successful integration of the best of these concepts is absolutely fundamental to the development, procurement, and use of highly survivable systems and networks.

Survivability of systems and networks is not an intrinsic low-level property of subsystems in the small. Instead, it is an *emergent property* — that is, a property that has meaning only in the context to which it relates. Emergent properties can be defined in terms of the concepts of their own layers of abstraction, but generally not in terms of individual components at lower layers. That is, an emergent property is a property that emerges as a result of the composition of lower-layer components. Simply composing a system or network out of its components provides no certainty whatever that the resulting whole will work as desired, even if the components themselves seem to behave properly. One of the most important challenges confronting us is to be able to derive the emergent properties of a system in the large from the properties of its components and from the manner in which they are integrated.

To satisfy the goals stated above, we take a strongly system-oriented approach. This approach

- Examines the survivability requirements and their interdependence with one another
- Recommends the development of specific infrastructural components that are currently missing or not commercially available
- Explores operational principles that can enhance survivability
- Characterizes architectures that can achieve critical survivability requirements

Recommendations

The following recommendations are considered further in Section 10.1. Specific directions for research and development are discussed in Section 10.2.

- We must establish prototypical requirements for survivability and its subtended properties that can be directly applied to system developments and procurements.
- We must develop new network and distributed system protocols in anticipation of the creation of highly survivable, secure, and reliable information infrastructures.
- We must establish a library of demonstrably sound robust procedures that enable trustworthy systems to be built out of less trustworthy components. This is the concept of generalized dependence, which we explore in this report.
- We must find ways to encourage system developers to increase the survivability, security, and reliability of their standard products.
- We must find ways to encourage commercial system developers to bring more of the good research and development results into the commercial marketplace.

- We must establish and consistently use sound cryptographic infrastructures for authentication, certificate authorities, and confidentiality.
- We must pursue more widespread development and use of nonproprietary open-source software and nonproprietary interfaces, and provide for mechanisms for trustworthy distribution of code – including robust mobile code.
- We must establish a collection of open-system architectural components that satisfy strong requirements for survivability and interoperability.
- We must pursue research and development relating to practical system composability that is also strongly based theoretically.
- We must refine and make practical the ongoing R&D efforts for monitoring, analyzing, and responding to system and network anomalies, and generalize them from merely intrusion-detection systems, so that they address a broad range of survivability-related threats, including reliability problems, fault-tolerance coverage failures, and classical network management.
- We must find ways to disseminate the concepts of this report widely. (See the appendix for some suggestions.)

Architecture and Implementation

The use of structure is particularly important in designing, implementing, and maintaining systems and networks. The combination of architectural principles and the use of good software engineering and system engineering practice can be extremely effective. In particular, it is vital to address the full range of survivability-relevant requirements from the outset; it is typically very difficult to make retrofits later. The notion of generalized dependence permits us to avoid needing total dependence on the correctness of certain other components – many of which have unknown trustworthiness, or are inherently suspect. This is increasingly important in highly distributed computing environments. The mobile-code paradigm offers many potential advantages in such environments, but it also requires some dramatic improvements in the robustness, security, and reliability of certain critical components.

Conclusions

It is a difficult course that we must follow. It is evidently a never-ending course, for a variety of reasons. As the requirements continue to be better understood, more is demanded. As technical improvements are introduced, new vulnerabilities are typically introduced. As technology continues to offer new functional opportunities, and as systems tend to operate closer to their technological limits, the vulnerabilities, threats, and risks are increased accordingly, requiring much greater care. Operational and administrative challenges are continually increasing. Also, our adversaries are becoming much more agile and are quite capable of becoming much more aggressive. As a consequence, much greater discipline is required to achieve the necessary goals. This report attempts to characterize what is needed in terms of increased awareness and approaches for the future.

Chapter 1

Introduction

> 1. *Out of clutter, find simplicity.*
> 2. *From discord, find harmony.*
> 3. *In the middle of difficulty lies opportunity.*
>
> Albert Einstein, three rules of work

1.1 Project Goals

The primary goal of this project is to significantly advance the state of the art in obtaining highly survivable systems and networks, whereby distributed systems and networks of systems are considered in their totality as systems of systems — rather than more conventional approaches that focus only on selected properties of certain subsystems or modules in isolation.

To accomplish that goal in this report, Chapter 2 addresses a broad spectrum of threats to survivability. Chapter 3 considers the overarching survivability requirements necessary to surmount those threats, and also considers the subordinate requirements on which survivability ultimately depends — including reliability, availability, security (confidentiality, integrity, defense against denials of service and other types of misuse), performance, in the presence of accidental and malicious actions and malfunctions of software and hardware. Chapter 4 then identifies fundamental deficiencies in the technology available today, and Chapter 5 makes recommendations for how to overcome those deficiencies. Subsequent chapters address guidelines for developing and rapidly configuring highly survivable systems and networks, including the presentation of generic classes of architectural structures and some specific types of systems. The appendix considers how the contents of this report might find their way into an educational curriculum.

Despite the quoted dictum of Albert Einstein at the beginning of this chapter, we observe that general-purpose systems and networks that must be highly survivable are not likely to be simple — unless they are seriously trivialized. The nature of the problems is intrinsically complex: experience shows that many vulnerabilities are commonplace, and not easy to avoid; the potential threats are very broadly based; complexity is often beyond the scope of a small closely knit development team; management is often unaware of the complexities and their implications. Consequently, the approach of this report is to confront the challenge in its full generality, rather than merely to carve out a simply manageable small subset. Remember the following quote, which is also very pithy:

> *Everything should be as simple as possible, but no simpler.*
> Albert Einstein

Recognizing the complexity inherent in satisfying any realistic set of survivability requirements, we have chosen to consider the very difficult fully general problem of achieving highly survivable systems and networks subject to the widest spectrum of threats. By tackling the general problem, we believe that much greater insight can be gained and that the resulting approaches can look farther into the future. In this sense, we believe that there is a significant opportunity in the face of the intrinsic difficulties.

1.2 Fundamental Concepts

Basic concepts are identified and define here that are used throughout the report, including survivability, security, reliability, performance, trustworthiness, dependability, assurance, mandatory policies, composition, and dependence. Section 1.3 introduces the notion of compromisibility.

1.2.1 Survivability

For the purposes of this report, *survivability* is the ability of a computer-communication system-based application to satisfy and to continue to satisfy certain critical requirements (e.g., specific requirements for security, reliability, real-time responsiveness, and correctness) in the face of adverse conditions. Survivability must be defined with respect to the set of adversities that are supposed to be withstood. Types of adversities might typically include hardware faults, software flaws, attacks on systems and networks perpetrated by malicious users, and electromagnetic interference.[1] Thus, we are seeking systems and networks that can prevent a wide range of systemic failures as well as penetrations and internal misuse, and can also in some sense tolerate additional failures or misuses that cannot be prevented.

As currently defined in practice, requirements in use today for survivable systems and networks typically fall far short of what is really needed. Even worse, the currently available operating systems and networks fall even farther short. Consequently, before attempting to discuss survivable systems, it is important to establish a comprehensive set of realistic requirements for survivability (as in Chapter 3). It is also desirable to identify fundamental gaps in what is currently available (as in Chapter 4).

Given a well-defined set of requirements, it is then important to define a family of reusable interoperable baseline system and network architectures that can demonstrably attain those requirements — with the goals of enhancing the procurement, development, configuration, assurance, evaluation, and operation of systems and networks with critical survivability requirements.

---

[1] As is often done for reliability, survivability could alternatively be defined as a probabilistic measure of how well the given requirements are satisfied. However, quantitative measures tend to be very misleading — especially when based on incorrect or unfounded assumptions. Thus, we prefer an unquantified definition.

A preliminary scoping of the general survivability problem was suggested by a 1993 report written for ARL, *Survivable Computer-Communication Systems: The Problem and Working Group Recommendations* [25]. That report outlines a comprehensive multifunctional set of realistic computer-communication survivability requirements and makes related recommendations applicable to U.S. Army and defense systems.[2] It assesses the vulnerabilities, threats, and risks associated with applications requiring survivable computer-communication systems. It discusses the requirements, and identifies various obstacles that must be overcome. It presents recommendations on specific directions for future research and development that would significantly aid in the development and operation of systems capable of meeting advanced requirements for survivability. It has proven to be useful to ARL as a baseline tutorial document for bringing Army personnel up to speed on system vulnerabilities and basic concepts of survivability. It remains timely. Some of its recommended research and development efforts have still not been carried out, and are revisited here.

The current technical approach is strongly motivated by a collection of highly disciplined system-engineering and software-engineering concepts that can add significantly to the generality and reusability of the results, as well as having specific applicability to Army developments. Above all, our approach stresses the importance of sound system and network architectures that seriously address the necessary survivability requirements. This approach entails several basic concepts that are considered in the following subsections.

1.2.2 Attributes of System Survivability

The following three bulleted items consider three types of infrastructures: (1) the critical national infrastructures, (2) information infrastructures such as the Internet, or whatever may replace it — a National Information Infrastructure, or a Global Information Infrastructure, or a Solar-System Information Infrastructure, or perhaps even the Intergalactic Information Infrastructure, and (3) underlying computer systems and networking software.

- **Survivability of the critical national infrastructures.** The President's Commission on Critical Infrastructure Protection (PCCIP) recently released a report [182] summarizing its recommendations relating to eight major critical national infrastructures: telecommunications; generation, transmission, and distribution of electric power; storage and distribution of gas and oil; water supplies; transportation; banking and finance; emergency services; and continuity of Government services. Perhaps most important from the present perspective is the recognition that very serious vulnerabilities and threats exist in these critical infrastructures. Perhaps equally important is the

---

[2] The working group consisted of Charles Meincke (Chairman, AMSRL-SL-EI), Anthony Barnes (AMSRL-SL-EI), Andrew Hollway (deceased, VAL FIO), Robert Hein (CECOM SWD), Bernard Newman (CECOM C3), John Preusse (CECOM C3), Barry Salis (CECOM C3 Systems), Phillip Dykstra (BRL/SECAD), Beth Manahan (ATZH-TH), Thomas Reardon (USAISC HQ), Joseph Stevenson (USAISC HQ), Laura Vaglia (SAA, INSCOM), Jeni Wilson-Galleher (AMC HQ), Dennis Steinauer (NIST), Marianne Swanson (NIST), and Peter Neumann — who did most of the writing. Organizational designations are given as of the date of the report's publication in 1993.

Commission's recognition that these critical infrastructures are closely interdependent and that they all depend on underlying computer-communication infrastructures.

- **Survivability of the computer-communication infrastructures on which the national infrastructures depend, such as the Internet and its eventual successors.** A comprehensive system- and network-wide set of realistic requirements is desired, encompassing security, reliability, fault tolerance, real-time performance, and any other attributes necessary for attaining adequate system and network survivability. From this set of requirements, it is possible to select those that are specifically relevant to any desired system. The Internet is seriously vulnerable to denial-of-service attacks, losses of confidentiality and integrity in transmission, and collapse of constituent nodes. In the future, critical applications are likely to demand alternative information infrastructures that provide greater survivability and its subtended requirements, with particular attention to reliability, availability, and security — cryptographically based confidentiality and integrity, protection against denials of service, and basically a completely new set of protocols designed with security in mind from the very beginning — including secure interoperability with other infrastructures. Such a national information infrastructure would be of enormous value to DoD and to the critical national infrastructures, and would also be valuable for electronic commerce.

- **Survivability of the underlying computer systems and communication systems.** Survivability of the computer-communication infrastructures depends on dependable operating system and network security, dependable system and network reliability, and dependable operational performance.

System attributes that are particularly relevant to the attainment of survivability include the following.

- **Security.** Security must encompass dependable protection against all relevant concerns, including confidentiality, integrity, availability despite attempted compromises, preventing denials of service, preventing and detecting misuse, providing timely responses to perceived threats, and reducing the consequences of unforeseen threats. It includes both system security (e.g., protecting systems and networks against tampering and other misuse) and information security (e.g., protecting data and programs against tampering and other misuse). It must anticipate all realistic threats, including misuse by insiders, penetrations by outsiders, accidental and intentional interference (e.g., electromagnetic), emanations, covert channels, inference, and aggregations. There is much more to security than merely providing confidentiality, integrity, and availability. All components that must be trusted in order to achieve adequate system behavior must actually be trustworthy. (The distinction between the two concepts is discussed in Section 1.2.3.)

- **Reliability and availability.** Reliability is generally defined as a measure of how well a system operates within its specifications. For, example, fault tolerance can enable a variety of alternatives, including real-time, fail-safe, fail-soft, fail-fast, and fail-secure modes of operation. Availability despite system failures must be tailored to a variety of specific needs, with different techniques used for different functionality, as appropriate. It must address unintentional and malicious changes in the operating environment, including those that result from power outages and power variations, earthquakes, floods, and other natural disasters. There should be no serious weak links that are vulnerable to perceived threats, and system design should be defensive enough that it also addresses some of the more serious unanticipated threats.

- **Performance.** Particularly in real-time systems, performance is a critical requirement. In some cases, adequate performance may be critical to the survivability of an enterprise or an application. On the other hand, in most cases, performance is itself dependent on survivability and availability — if a system is not survivable, adequate performance cannot be achieved. To avoid this apparent interdependence loop, it would be possible to redefine performance requirements as being meaningfully specified only when the relevant systems are available. However, this seems to be a cop-out, because of the need to ensure adequate performance that is itself critical to survivability.

What is immediately obvious is that close interrelationships exist among the various requirements. For example, consider the various forms of availability. Availability is clearly a security requirement in defending against malicious attacks. It is clearly a reliability requirement in defending against hardware malfunctions, unanticipated software flaws, environmental causes, and acts of God. It is also a performance issue, in that adequate availability is essential to maintaining adequate performance (and conversely, adequate performance can be essential to maintaining adequate availability, as noted above).

Whereas it is conceptually possible to consider these different manifestations of availability separately as requirements, this is very misleading — because they are closely coupled in the design and implementation of real systems and networks. As a consequence, we stress the notion of architectures that address these seemingly different requirements in an integrated way that permits the realization of different requirements within a common structure. This is pursued further in Section 3.1.

1.2.3 Trustworthiness, Dependability, and Assurance

Fundamental to this report are the notions of trustworthiness, dependability, and assurance.

- **Trustworthiness versus trust.** In the present context, *trustworthiness* is a measure of how extensively a given module, system, network, or other entity deserves to be trusted to satisfy its stated requirements when confronted with arbitrary threats. In the security community, trustworthiness is roughly equivalent to what is called "dependability" in the fault-tolerance community, although dependability was originally relevant primarily to the threats intended to be covered by fault tolerance, and did not encompass what we refer to here as trustworthiness. (In the present context, *assurance* — considered below — relates to the certainty with which trustworthiness or dependability can be believed, for example, through the use of testing and formal analyses.) In this report, the notion of trustworthiness encompasses all aspects of survivability, including the full spectrum of threats to survivability and its subtended requirements.

Trustworthiness tends to be used in situations where the requirements are particularly critical, that is, where dependence on the trustworthiness of specific entities is crucial to the overall behavior of a system or network in the large — particularly with respect to survivability, security, and reliability. In the fault-tolerance community, dependability tends to be a measure of how well the specified fault-tolerance requirements are met.

A careful distinction is made here between trust and trustworthiness. Trust is something you attribute to a system entity, whether that entity is trustworthy or not. A trustworthy entity is one that deserves to be trusted.

In general usage in the literature, a trusted system is one that must be trusted in order for applications using the system to behave properly. Ideally, trusted systems should be trustworthy, although that is often not the case. For example, the notion of trusted computing bases (Section 8.2) is really concerned with trustworthiness of components that, because of their functionality, have to be trusted — and that therefore must be trustworthy.

- **Dependability and assurance.** In this report, we refer only loosely to *dependability,* as the extent to which a given requirement is perceived to be satisfied, particularly by the implementation. *Assurance* is then the credibility that can be given to specific statements of dependability and trustworthiness. Thus, for example, we might rather specifically refer to the assurance of the dependability of trustworthiness within a particular system design or its implementation, although in general we are primarily concerned with trustworthiness itself and avoid the circumlocution. The reader will not suffer by assuming that the assurance of trustworthiness and the predictability of dependability (e.g., [283]) are indeed the same concept.

  The foregoing concepts — survivability, security, reliability, and performance — need to be implemented in such a way that the desired properties can be achieved dependably. Defensive measures include establishment of appropriate requirements, good system design that is consistent with the requirements, good system development and coding practice including the use of modern software engineering and sound programming languages and demonstrations that implementations are consistent with their designs, and operational procedures that maintain the integrity of design and implementation despite ongoing debugging and maintenance.

- **Assurance and analytic dependability.** Overall system and network survivability should be predictably demonstrable by methods other than simply testing. For example, formal methods could be used to demonstrate the adequacy of the requirements, the consistency of the specifications with those requirements, and the consistency of the implementation with the specifications, at different layers of abstraction, and the consistency of one layer with another. Alternatively, other constructive analytic arguments could measure the assurance with which a system architecture might survive specific types of threats, even despite unanticipated events. Various approaches to assurance exist, including formal methods discussed in Section 5.8.

Various other attributes are also highly desirable in ensuring dependable survivability.

- **Subsystem composability.** Subsystems and components should be designed and implemented to enhance the ease with which they can be integrated together without adversely affecting survivability. Composability is considered further in Section 5.7.

- **Interoperability.** Interoperability should be easily attainable, across different networks, systems, subsystems, and applications. Various platforms should be accommodated, including mainframes, minis, workstations, personal computers, and combinations thereof. Firewalls and other controls necessary for security should not unduly impede interoperability where authorized. Relevant standards should be respected, but inadequate standards must be replaced, upgraded, or explicitly ignored.

- **Scalability.** The above-mentioned concepts must be capable of adapting to a range of operations, from local operation to widely dispersed systems, from a few users to a considerable community of users, from a closed community to an open-system environment. Where single approaches are not applicable, the parameterized configuration should permit adaptation according to the specific requirements.

- **Abstraction.** At each of various layers of abstraction, implementations must be properly (e.g., survivably, securely, and reliably) encapsulated, with appropriate information hiding and high-integrity implementation. System interfaces and programming languages must provide suitable abstractions, and must be nonbypassable.

- **Prevention, detection, toleration, and reaction.** Neither prevention nor detection is adequate by itself. An appropriate balance must be found between the two, utilizing constructive design techniques as well as proactive detection of events that may be suspicious with respect to the requirements for survivability, security, reliability, and so forth. In addition, toleration of compromises must be planned in system design, for cases in which prevention does not succeed, irrespective of whether detection succeeds. Traditional fault tolerance is intended to combat reliability-related threats, whereas its counterpart in security-related threats must be to withstand to whatever degree practicable malicious attacks that cannot be prevented. Finally, speedy reaction is necessary when something adverse has been detected.

- **Distributed protection domains.** The concept of trusted computing bases must be respected, despite its inherent limitations. Separation of privileges and allocation of least privilege should be observed throughout, with clearly delineated protection domains, carefully defined and enforced hierarchical layering, and logical compartments. Although there may be applications for which centralized mechanisms may be functionally appropriate, even those mechanisms must be able to be physically or logically distributed in order to attain high availability. (For example, having only a single authentication server is ill advised in distributed environments, particularly in the presence of many users or resources.)

- **Dependencies and interrelationships.** In any distributed system architecture, whether hierarchically layered or highly interrelational, it is undesirable to have to depend on inherently less trustworthy components — with respect to security, reliability, and availability. If such adverse dependence is essential to minimize or control harmful effects, it must be demonstrably not in conflict with the desired requirements. For example, deadlocks can be caused accidentally or intentionally, and can result in denials of service unless the system architecture takes explicit measures to avoid them. Devastating consequences can result from dependence on untrustworthy components that are nevertheless ill-advisedly trusted. It is also desirable to identify and analyze the interactions among the different requirements — with respect to the requirements themselves, as well as with respect to conflicts that may arise in the design and implementation. If potentially conflicting consequences can arise, the priorities necessary to resolve those conflicts should be established in advance, insofar as possible.

- **Operational practice.** The best design and implementation can be totally compromised by bad administrative and operational practice. Sound configuration control is absolutely essential as an integral part of survivability. (For example, see [351].)

Whereas we have chosen a framework in which survivability depends on security, reliability, and performance attributes (for example), manifestations of survivability, security, and reliability exist at many different layers of abstraction. Although the survivability of an enterprise may depend on the underlying security and reliability, the security and reliability at a particular layer may in turn depend to some extent on the survivability of a lower layer. For example, the survivability of each of the eight critical national infrastructures considered by the PCCIP depends to some extent on the survivability and other attributes of the underlying computer-communication infrastructures. Similarly, the survivability of a given computer-communication infrastructure may typically depend to considerable extent on the survivability of the electric power and telecommunications infrastructures. In part, this is a consequence of the fact that the definitions used here are (necessarily) somewhat overlapping; in part, it is also a recognition of the fact that each abstract layer has its own set of requirements that must be translated into subrequirements at lower layers.

One of the primary goals of the present work is to identify the ways in which the various properties and their enforcing implementations depend on one another, at various layers of abstraction and across different abstractions at given layers.

This report in no way attempts to be a definitive self-contained treatise on everything that needs to be known to procurers and developers of highly survivable systems. Rather, it attempts to identify and use constructively some of the fundamental concepts upon which such systems can be produced. Extensive further background on computer system trustworthiness can be found in National Research Council reports, *Computers at Risk* [67] and the more recent *Trust in Cyberspace* [312]. Two valuable volumes on cryptography's role in trustworthy systems and networks are the NRC CRISIS report [79] and Bruce Schneier's Applied Cryptography [313].

1.2.4 Generalized Composition

Research efforts have typically considered simple compositions of modules, such as unidirectional serial connections or perhaps call-and-return semantics. (Section 5.7 discusses some of these.) However, the existing research is far from realistic.

The concept of *generalized composition* [229] used here includes composition of subsystems with mutual feedback, hierarchical layering in which a collection of modules forms a layer that can be used by higher layers as in the Provably Secure Operating System (PSOS) [94, 224, 225, 235], layering achieved through program modularity [42], and networked connections involving client-server architectures, gateways, unidirectional and bidirectional firewalls and guards, encryption, and other components. Relevant approaches include [334].

In this project, we consider generalized composition as it relates to the composed subsystems. We believe that this approach to composition is more appropriate to the intended large-scale distributed and networked architectures than the primarily theoretical contemporary work on model composition and policy composition (although that work is logically subsumed under the present approach).

1.2.5 Generalized Dependence

In 1974, Parnas [252] characterized a variety of `depends upon` relations. An important such relation is Parnas's `depends upon for its correctness`, whereby a given component is said to depend upon another component in the sense that if the latter component does not meet its requirements, then the former may not meet its requirements. Neumann [229] has revisited the notion of dependence, making a distinction between the Parnas relation `depends upon for correctness` and a generalized sense of dependence in which greater trustworthiness can be achieved despite the presence of less trustworthy components, thereby avoiding having to depend completely on components of unknown or uncertain trustworthiness. To avoid having to say "depends upon in the sense of generalized dependence", we abbreviate that generalized relation as simply `depends on`.

The following enumeration gives various paradigms under which trustworthiness can actually be enhanced, providing examples of how the generalized dependence relation `depends-on` differs from the conventional `depends-upon` relation. In each of these cases, the resulting trustworthiness tends to be greater than that of the constituent components. The list is surprisingly long, and may help to illustrate the power of the notion of generalized dependence. (Although particular mechanisms may fall into multiple types, these types are intended to represent the diverse nature of mechanisms having the characteristics of generalized dependence.)

1. The use of error-correcting codes (e.g., [114]) that can enable correct communications despite certain tolerable patterns of errors (e.g., random, asymmetric as in bit-dropping only, bursty, or otherwise correlated), in block communications or even in variable-length or sequential encoding schemes, as long as any required redundancy does not cause the available channel capacity to be exceeded (following the guidance of Shannon's information theory)

2. The early work of John von Neumann [346] and of Ed Moore and Claude Shannon [201], who showed how reliable subsystems in general (von Neumann) and reliable relay circuits in particular (Moore-Shannon) can be built out of unreliable components — as long as the probability of failure of each component is not precisely one-half and as long as those probabilities are independent from one another

3. Self-synchronizing techniques that result in rapid resynchronization following nontolerated errors that cause loss of synchronization, including intrinsic resynchronizability

of variable-length and sequential coding schemes — in some cases even without the necessity of adding redundancy [218, 219, 220] in cases of variable-length [126] and information-lossless [127] sequential encoding systems

4. Traditional fault-tolerance algorithms and system concepts that can tolerate certain specific types of component or subsystem failures as a result of constructive use of redundancy [17, 75, 135, 174, 192, 204, 350] — although failures beyond the coverage of the fault tolerance may result in unspecified failure modes

5. Alternative-computation architectural structures, which can achieve satisfactory but nonequivalent results (with possibly degraded performance), despite failures of hardware and software components, such as the Newcastle Recovery Blocks approach [16, 17, 125] and failure modes that exceed planned fault coverage

6. Alternative-routing schemes in packet-switched networks, which can attain good performance and eventual communications despite major outages among intermediate nodes and disturbances in communications media (as in the ARPAnet routing protocols)

7. Robust synchronization algorithms, such as hierarchically prioritized locking strategies [88], two-phase commitments, nonblocking atomic commitments [285], and fulfillment transactions [191] such as fair-exchange protocols guaranteeing that payment is made if and only if goods have been delivered

8. Byzantine fault-tolerant systems that can withstand Byzantine fault modes [154, 303, 310], whereby successful operation is possible despite the arbitrary and completely unpredictable behavior (maliciously or accidentally) of up to some ratio of its component subsystems (e.g., k out of $3k + 1$), with no assumptions regarding individual failure modes of the component subsystems

9. Byzantine network-layer protocols [266]

10. Micali's fair public-key cryptographic schemes [195], in which different parties must cooperate with the simultaneous presentation of multiple keys — allowing cryptographically based operations to require the presence of multiple authorities

11. Threshold multikey-cryptography schemes, in which at least k out of n keys are required (for conventional symmetric-key decryption, or for authentication, or for escrowed retrieval) — for example, a Byzantine digital-signature system [85] and a Byzantine key-escrow system [288] that can function successfully despite the presence of some parties that may be untrustworthy or unavailable, as well as a signature scheme that can function correctly despite the presence of malicious verifiers [267]

12. Byzantine-style authentication protocols that can work properly despite untrustworthy user workstations, compromised authentication servers, and other questionable components (see Chapter 8)

13. Constructive use of kernels and "trusted" computing bases to achieve nonsubvertible application properties, such as in SeaView, which demonstrated how a multilevel-secure DBMS can be implemented on top of an MLS kernel — with absolutely no requirement for MLS trustworthiness in the DBMS [83, 176, 178]

14. Multilateral mechanisms enforcing policies of mutual suspicion, with the ability to operate correctly despite a lack of trust among the various parties [316]

15. Interposition of trustworthy firewalls and guards that mediate between regions of unequal trustworthiness — for example, ensuring that sensitive information does not leak out and that Trojan horses and other harmful effects do not sneak in, despite the presence of untrustworthy subsystems or mutually suspicious adversaries

16. Use of run-time checks to prevent or mediate execution in questionable circumstances (e.g., embedded in the base programs or in application programs, as in the cases of bounds checks and consistency checks)

17. Addition of wrappers that enhance survivability, security, or reliability, or otherwise compensate for deficient components — such as adding a "trusted path" to an inherently untrustworthy system, enabling monitoring of otherwise unmonitorable functionality, or providing compatibility of wrapped legacy programs with other programs

18. Use of integrity checks, such as cryptographic checksums that ensure the constancy of data and programs, and proof-carrying code [213] that can enable the detection of unexpected alterations to systems or data

19. Object-oriented, domain-enforcement, and access-control techniques that effectively mediate or otherwise modify the intent of certain attempted operations, depending on the execution context [94, 235, 316] — for example, the confined environment of the Java Virtual Machine [105, 107] and related work on formal specification [82, 103] for the analysis of the security of such environments

20. Use of real-time analysis techniques such as anomaly and misuse detection to diagnose live threats and respond accordingly, capable of dynamically altering system and network configurations based on perceived threats (e.g., [275, 276])

Each of these paradigms demonstrates techniques whereby trustworthiness can be enhanced above what can be expected of the constituent subsystems or transmission media. By generalizing the notions of dependence and trustworthiness, and judicious use of some of these techniques, we seek to provide a unifying framework for the development of survivable systems.

Dependence on components and information of unknown trustworthiness is a particularly serious potential problem. (See Sections 2.1.1 and 2.1.2.)

Dependable clocks (Byzantine or otherwise) provide a particularly interesting challenge. Lincoln, Rushby, and others [170] provide an elegant detailed example of generalized dependence. They have analyzed a three-layered model consisting of (1) clock synchronization [301], (2) Byzantine agreement [168, 169], and (3) diagnosis and removal of faulty components [169]. They also exhibit formal verifications for a variety of hybrid algorithms [169] that can greatly increase the coverage of misbehaving components. This three-layered integration of separate models and proofs is of considerable practical interest, as well as illustrative of forefront uses of formal methods.

A recent example of generalized dependence relating to clock drift is given by Fetzer and Cristian [96] in developing fault-tolerant hardware clocks out of commercial off-the-shelf (COTS) components, at least one of which is a GPS receiver. A formal analysis of a time-triggered clock synchronization approach is given by [270].

The basic approach of this project considers within a common framework many different generalized-dependence mechanisms that are capable of enhancing trustworthiness, enabling the resulting functionality to be inherently more trustworthy than otherwise might be warranted by consideration of only its constituent components.

1.2.6 Generalized Survivability

Ultimately, overall system survivability may depend on (in the sense of generalized dependence noted above) the security, integrity, reliability, availability, and performance characteristics of certain critical portions of the underlying computer-communication infrastructures. In this report, we use the term *generalized survivability* to refer to survivability in the context of generalized dependence. *Compromises from outside, from within,* or *from below* (see Section 1.3 and [228, 229, 241]), whether malicious or not, can subvert survivability unless prevented or ameliorated by the architecture, its implementation, and the operational practice. Unfortunately, compromises from outside (e.g., externally, originating from higher layers of abstraction or from other entities at the same layer of abstraction, or from supposedly security-neutral applications) often can lead to compromises from within (affecting the implementation of a particular mechanism) or from below (subverting a mechanism by tampering with its underlying dependent components). One of the fundamental challenges addressed here is to be able to design, implement, and operate survivable systems despite the presence of components, information, and individuals of unknown trustworthiness — as well as saboteurs (e.g., cyberterrorism [273]), and thereby to prevent, defend against, or at least detect attempted compromises from outside, within, or below. This is in essence what we mean by generalized survivability — survivability in the context of generalized dependence on potentially unknown entities. For example, a particularly difficult challenge is to ensure that the embeddings of sound cryptographic algorithms cannot be compromised because of inherent weaknesses in the underlying computer-communication infrastructures (e.g., hardware, microcode, operating systems, database management, and networking) — as discussed in [227].

Generalized survivability is an *emergent* property of the overall systems and networks. That is, it is not definable and analyzable in the small, because it is the consequence of the subtended functionality; it must be considered in the large. In other words, it is not a property that can be identified with any of the constituent components. Ideally, it should be derivable in terms of properties of the constituent functionality on which it depends, as described in the 1970s work of Robinson and Levitt [292] on the SRI Hierarchical Development Methodology (which dealt with emergent properties without so naming them) as part of the PSOS effort.

Stephanie Forrest in her introduction to the 1991 CNLS proceedings [98], Nancy Leveson [163], Heather Hinton [118, 119], Zakinthinos and Lee [354], and D.K. Prasad [277] provide some background on emergent properties; Zakinthinos and Lee define an emergent property as one that its constituent components do not satisfy. Prasad draws on measurement theory and decision analysis [278] to show that such properties are not compositional and also that such properties are not 'absolute' — different stakeholders may have different ideas about the meaning of the property. Her thesis work also presents the method of multi-criteria decision making (in a specific framework) as an approach for the measurement (on a sound theoretical basis) of such properties. Hinton [119] observes that undesirable emergent behavior is often the result of incomplete specification, and can be formally analyzed.

1.2.7 Mandatory Policies for Security, Integrity, and Availability

The notions of multilevel security [28, 29, 30, 31, 32], multilevel integrity [38, 39], and multilevel availability [241] characterize hierarchical mandatory policies for confidentiality, integrity, and availability, respectively. In *multilevel security* (MLS), information is not permitted to flow from one entity to another entity that has been assigned a lower security level. In *multilevel integrity* (MLI), no entity is permitted to depend upon an entity that has been assigned a lower integrity level. In *multilevel availability* (MLA), no entity is permitted to depend on an entity that has been assigned a lower availability level.

Although it has been the subject of considerable research in security policies and kernelized system architectures, and highly touted by the Department of Defense (see Chapter 6), multilevel security has remained very difficult to achieve in realistic systems and networks. This is due to many factors, including inadequacies in the DoD criteria, an unwillingness of commercial system providers to develop systems, and an unwillingness of non-DoD system acquirers to consider such systems. Alternative architectures are considered in Chapter 8.

Strict multilevel integrity is thought to be awkward to enforce in practical systems, because high-integrity users and processes often depend on editors, compilers, library routines, device drivers, and so on, that are typically not necessarily trustworthy and therefore dangerous to depend upon. However, that is precisely the fundamental integrity problem in most system architectures. The implicit web of trust should force those utility functions to be at least as trustworthy with respect to integrity, because they must all be considered within the perimeter of trustworthiness. The notion of generalized dependence is one way of working within that constraint without either sacrificing the power of the basic concepts or of introducing new vulnerabilities that result from informal deviations from strict interpretations.

1.2.8 Multilevel Survivability

In this report, we consider the conceptual application of this kind of mandatory basis to generalized survivability. This leads to lattice-based mandatory policies for multilevel generalized survivability that directly imitate the MLS, MLI, and MLA policies. For simplicity, we refer to this policy as simply *multilevel survivability* (MLX). In an oversimplified formulation of the multilevel survivability policy, no system or network entity is allowed to depend on an entity that has been assigned a lower survivability level (unless an explicit generalized-dependence mechanism is established that permits the use of mechanisms of lower trustworthiness, as illustrated in Section 1.2.5). These concepts are thus extended rather simply in this report to accommodate generalized dependence.

For descriptive simplicity, we ignore the possible existence of compartments in each of these policies (MLS, MLI, MLA, and MLX), and describe the policies only in terms of levels. However, the presence of compartments is nevertheless assumed in all cases (unless otherwise stated explicitly). Thus, the ordering on the levels and compartments generates a mathematical lattice in each instance. When we refer to mandatory policies in this context, we imply lattice-based policies rather than just completely ordered levels (without compartments).

In the absence of generalized dependence, strict MLX ordering would most likely suffer the same kind of problems that arise in the practical use of strict MLI — namely, the realization that enormous portions of any given distributed system must be of high integrity and high survivability. The notion of generalized dependence therefore allows the strict partial ordering to be relaxed locally whenever it is possible to achieve greater trustworthiness out of less trustworthy components, as illustrated in Section 1.2.5 — without relaxing it in the large.

This initial discussion represents a first approximation to what is actually needed. In Section 8, we address the possible conflicts among the subrequirements of survivability in the context of generalized dependence.

1.3 Compromisibility and Noncompromisibility

To illustrate the importance of dependence on properties of underlying abstractions, consider the necessity of depending on a life-critical system for the protection of human safety.[3] In such a system, safety ultimately **depends upon** the confidentiality, integrity, and availability of both the system and its data. It may also depend on information survivability. It may further **depend upon** component and system reliability, and on real-time performance. It also usually **depends upon** the correctness of much of the application code. In the sense that each layer in a hierarchical system design **depends upon** the properties of the lower layers, the way in which trusted computing bases are layered becomes important for developing dependably safe systems — particularly in those cases in which the generalized **depends on** relation can be used more appropriately instead of **depends upon** to accommodate an implementation based on less trustworthy components.

The same dependence situation is true of secure systems, in which each layer in the abstraction hierarchy (e.g., consisting of a kernel, a trusted computing base for primitive security, databases, application software, and user software) must enforce some set of security properties. The properties may differ from layer to layer, and various trustworthy mechanisms may exist at each layer, but the properties at a particular layer are derivable from lower-layer properties.

In the security context, many notions of compromise exist. For example, compromise might entail accessing supposedly restricted data, inserting unvalidated code into a trusted environment, altering existing user data or operating-system parameters, causing a denial of service, finding an escape from a highly restricted menu interface, or installing or modifying a rule in a rule-base that results in subversion of an expert system.

---

[3] This section is adapted from [229].

There is an important distinction between having to depend on lower-layer functionality (whether it is trustworthy or not) and having some meaningful assurance that the lower-layer functionality is actually noncompromisible under a wide range of actual threats. Noncompromisibility is particularly important with respect to security, safety, and reliability.

Potentially, a supposedly sound system could be rendered unsound in any of three basic situations:

- **Compromise from outside** (intuitively, above or laterally – from elsewhere at the same abstraction layer)
- **Compromise from within** (intuitively, inside a component or layer)
- **Compromise from below** (intuitively, underneath)

Each of these situations could be caused intentionally, but could also happen accidentally. (For descriptive simplicity, a *user* may be a person, a process, an agent, a subsystem, another system, or any other computer-related entity.)

- Compromise from outside typically originates from an access point that is nominally external to the component being compromised.
  - In cases of purposeful compromise from outside, the perpetration is typically that of a completely unprivileged user or a partially privileged user who gains access to perpetrate a further compromise. In general, no authorization is required, possibly because of an exploitable flaw in the standard interface; in some cases authorization may be bypassed.
  - In cases of accidental compromise from outside, the compromise may result from an inadvertent program error in a higher layer that somehow affects a lower layer, or in another module at the same layer for which there is inadequate isolation.
- Compromise from within typically originates inside a particular component that is compromised, existing at a given layer of abstraction.
  - In cases of purposeful compromise from within, the perpetration is performed by a user or program that has somehow gained access (with or without authorization) to the internals of a component, such as privileged maintenance access to a database management system, network controller, or automatic teller machine. The component could be compromised by an authorized user who is misusing privileges, or by a penetrator; once a perpetrator has gained access to the component internals, such a distinction may be academic. Thus, compromise from within may follow from a compromise from outside that enables a subsequent penetration.
  - In cases of accidental compromise from within, the compromise could involve flaws or malfunctions associated with the particular system component.

- Compromise from below is initiated at a lower layer of abstraction than the layer at which the compromise of a given component occurs. Compromise from below may result from malicious action or accidental failure of an underlying mechanism on which the particular component depends. It typically affects the particular component by altering the state of lower-layer functionality, or in some cases merely by gaining access to information in a lower-layer abstraction and using that information in some unexpected way. This is roughly equivalent in meaning to *subversion.*
  - In cases of purposeful compromise from below, the perpetration is performed by a user who has somehow gained access (with or without authorization) to layers of abstraction underlying a particular component that is being compromised, which can then be undermined without attacking the component itself. Examples include (1) obtaining the unencrypted form of an encrypted message by reading a temporary file in storage, (2) finding an occurrence of a particular word in a restricted database to which access is not permitted by scanning the disk on which that database is stored, and (3) editing the raw text of an enqueued mail message after it is released by a user but before it is actually sent out by the mailer. Thus, compromise from below may follow from a compromise from outside or compromise from within that enables a subsequent penetration to the lower-layer mechanisms.
  - Cases of accidental compromise from below often involve the results of flaws or malfunctions at lower layers of abstraction that in some way alter or otherwise affect the expected behavior of the particular component. For example, consider a rather dramatic error that occurred in the early 1960s in the MIT Compatible Time-Sharing System (CTSS). The entire unencrypted file of user passwords was printed out as the message of the day for each new user login. This resulted from a shortsighted naming convention in the context editor being used at the same time by two different operators in a shared system directory [228]. Two temporary file names were the same for all invocations of the editor, and the temporary files became interchanged. Notable examples of hardware flaws include the Pentium floating divide flaw (discussed in the Risks Forum in RISKS-16.57-59,61,66,67,69,71,72,81, for example) and security-relevant flaws in other processors (e.g., [322]).

The distinctions among these three modes tend to disappear in systems that are not well structured, in which inside and outside are indistinguishable (as in systems with only one protection state), or in which outside and below are merged (as in flat systems that have no concept of hierarchy). In addition, compromises from outside may subsequently enable compromises from within, and compromises from outside or within may subsequently enable compromises from below.

Certain attack modes may occur in any of these forms of compromise. For example, consider the following Trojan-horse perpetrations, which can take place in each form.

- Compromise from outside: a letter bomb (e.g., electronic mail or Word macro virus) that when read or interpreted can result in unanticipated executions, or a spoofing attack that piggybacks on a line or replays a message

- Compromise from within: a surreptitious code patch that maintains a hidden trickle file of sensitive information within the program data
- Compromise from below: a wiretap implanted inside a telephone switch, or Ken Thompson's now-classical object-code modification of the C compiler that permitted a trapdoor routine to be planted in the login [335] (whereby it becomes clear that system security also depends upon the compiler). Thompson's Trojan horse was inserted into the *object* code of the C compiler (with no change in the source of the C compiler), lurking until the next recompilation of the login routine, when it created a trapdoor in the *object* code of the login routine (with no change to the source code of the login routine). The Trojan horse placed in the compiler was capable of reinserting itself into the object code of successive recompilations of the compiler itself, and thus was itself survivable! This suggests that compilers have some special problems of their own, as considered in Section 5.9.

Table 1.1 summarizes some properties whose nonsatisfaction could potentially compromise system behavior, by compromising confidentiality, integrity, availability, real-time performance, or correctness of the application code, either accidentally or intentionally. To illustrate such compromises, the table also indicates possible compromises — whether they involve modification (tampering) or not — that can occur from outside, from within, or from below, for each representative layer of abstraction. The distinctions are not always precise: a penetrator may compromise from outside, but once having penetrated, is then in position to compromise from below or from within. Thus, one type of compromise may be used to enable another. For this reason, the table characterizes only the primary modes of compromise. For example, a user entering through a resource access control package such as RACF or CA-TopSecret, or through a superuser mechanism, and gaining apparently legitimate access to the underlying operating system may then be able to undermine both operating-system integrity (compromise from within) and database integrity (compromise from below if through the operating system), even though the original compromise is from outside. Similarly, a software implementation of an encryption algorithm or of a cryptographic check sum used as an integrity seal can be compromised by someone gaining access to the unencrypted information in memory or to the encryption mechanism itself, at a lower layer of abstraction. A user exploiting an Internet Protocol router vulnerability may initially be able to compromise a system from within the logical layer of its networking software, but subsequently may create further compromises from outside or below. The Thompson compiler Trojan horse is a particularly interesting case, because it may not normally be thought of as compromise from below if the compiler is not understood to be something that is depended upon for its correct behavior. Indeed, it is a very bad policy to use an untrustworthy compiler to generate an operating system, and therefore the compiler must be considered "below".

From the table, we observe that a system may be inherently compromisible, in a variety of ways. The purpose of system design is not to make the system completely noncompromisible (which is impossible), but rather to provide some assurance that the most likely and most devastating compromises are properly addressed by the design, and — if compromises do occur — to be able to determine the causes and effects, to limit the negative consequences,

Table 1.1: Illustrative Compromises

Layer of abstraction	**Compromise from outside;** Needs exogirding	**Compromise from within;** Needs endogirding	**Compromise from below;** Needs undergirding
Outside environment		Acts of God, earthquakes, lightning, etc.	Chernobyl-like disasters caused by users or operators
User	Masqueraders	Accidental mistakes Intentional misuse	Application system outage or service denial
Application	Penetrations of application integrity	Programming errors in application code	Application (e.g., DBMS) undermined within operating systems (OSs)
Middleware	Penetration of Web and DBMS servers	Trojan horsing of Web and DBMS servers	Subversion of middleware from OS or network operations
Networking	Penetration of routers, firewalls Denials of service	Trojan horsing of network software	Capture of crypto keys within the OS Exploitation of lower protocol layers
Operating system	Penetrations of OS by unauthorized users	Flawed OS software Trojan-horsed OS Tampering by privileged processes	OS undermined from within hardware: faults exceeding fault tolerance; hardware flaws or sabotage
Hardware	Externally generated electromagnetic or other interference External power-utility glitches	Bad hardware design and implementation Hardware Trojan horses Unrecoverable faults Internal interference	Internal power irregularities
Inside environment	Malicious or accidental acts	Internal power supplies, tripped breakers, UPS/battery failures	

and to take appropriate actions. Thus, it is desirable to provide underlying mechanisms that are inherently difficult to compromise, and to build consistently on those mechanisms. On the other hand, in the presence of underlying mechanisms that are inherently compromisible, it may still be possible to use Byzantine-like strategies to make the higher-layer mechanisms less compromisible. However, flaws that permit compromise of the underlying layers are inherently risky unless the effects of such compromises can be strictly contained.

1.4 Defenses Against Compromises

Protection against the three forms of compromise noted in Section 1.3 — compromise from outside, compromise from within, and compromise from below — are referred to in this report as *exogirding, endogirding,* and *undergirding,* respectively – that is, providing outside barrier defenses, internal defenses, and defenses that protect underlying mechanisms, respectively.[4] In general, all three approaches are necessary. Various approaches are considered in Chapters 5, 8, and 9. For the purposes of this chapter, just a few illustrative examples are given here, relating to a few of the layers of abstraction shown in Table 1.1. As indicated by this summary, some of the techniques are quite different from one case to another, although other techniques are more generically applicable.

- **Exogirding:** Domain architectures protecting systems from their users, firewalls protecting one system from another, authentication mechanisms preventing penetrations at many different layers, use of encryption for confidentiality and integrity of information transmission, electromagnetic shielding
- **Endogirding:** compile-time and run-time checks such as bounds checks and type checks to minimize the effects of errant programs, fault tolerance, integrity checks to prevent and detect Trojan horses, use of encryption for confidentiality and integrity of stored information, monitoring of program behavior to detect misuse or aberrant system operation
- **Undergirding:** use of secure kernels to prevent higher-layer compromises, trustworthy operating systems and high-integrity hardware to support critical software functionality, use of special-purpose hardware (e.g., co-processors and cryptographic engines) to aid less trustworthy higher-layer systems

1.5 Sources of Risks

Some of the many stages of system development and use during which risks may arise are listed below, along with a few examples of what might go wrong (and, in most cases, what has gone wrong in the past). This list summarizes some of the main threats. Section 1.6

---

[4] In this usage, "undergirding" has its natural-language meaning; "exogirding" takes on the primary meaning of "girding", to avoid confusion with its other meanings; "endogirding" is introduced as an intermediate concept — neither outside nor below, but rather internal.

gives examples of specific illustrative cases.

Problems in the system development process involve people at each stage, and are illustrated by the following examples:

- **System conceptualization**: inappropriate application of the technology when the risks were actually too great, and avoidance of computerization when it would have been essential
- **Requirements definition**: erroneous, incomplete, and inconsistent requirements
- **Models**: false assumptions about the physical world, the operating environment, and human behavior
- **System design**: fundamental misconceptions and design flaws
- **Implementation**: program bugs, omissions, and Trojan horses causing unanticipated effects
- **Support systems**: poor programming languages, faulty compilers and debuggers, and misleading development tools whose use might permit the development of weak systems
- **Testing and verification**: incomplete testing, erroneous verification
- **Evolution**: sloppy maintenance, misconceived system upgrades, introduction of new flaws in attempts to fix old flaws
- **Decommission**: premature removal of a primary or backup facility, hidden dependence on an old version that is no longer available but that is required (e.g., for compatibility)
- **Stagnation**: necessary expansion of a system beyond its initial requirements that cannot be accommodated (e.g., because software bloat, loss of key personnel, or hardware unavailability makes upgrades and retrofits infeasible)

Problems in system operation and use involve people and external factors, and are illustrated by the following examples:

- **Hardware malfunction**, due to
  - Environmental factors such as lightning, earthquakes, extreme temperatures, electromagnetic[5] and other interference including cosmic radiation and sunspot activity, animals (sharks, rats, and squirrels are included in the case histories, for example) and many natural disasters

---

[5] For recent House testimony on some of the risks of RF interference, see *Radio Frequency Weapons and Proliferation: Potential Impact on the Economy,* Joint Economic Committee Hearing. 25 February 1998 (`http://www.house.gov/jec/hearings/02-25-8h.htm`). Systems developed in the former Soviet Union were previously discussed by General Schweitzer (`http://jya.com/rfw-jec.htm`).

  - Loss of electrical power
  - Component malfunction: aging, transient behavior, or inadequate design

- **Software misbehavior**: for example, due to problems in the system development process, as noted above

- **Human behavior in system use**, whether in system operators, administrators, staff, users, or unsuspecting bystanders, for example, in

  - Installation: improper configuration, incompatible versions, erroneous parameter settings, or linkage errors
  - Misuse of the overall environment or the computer systems, including
    - Unintentional misuse (including untimely use): entry of improper inputs, misinterpretation of outputs, or execution of the wrong function
    - Intentional misuse: penetration by unauthorized users, misuse by authorized users, insertion of Trojan horses, or fraud (Section 2.1)

The last subcategory — intentional misuse — represents a particular worrisome area of concern and is considered in Section 2.1.

1.6 Some Relevant Case Histories

We consider here just a few illustrative problems that have been encountered in the past, suggesting the rather pervasive nature of the survivability problem — with many diverse causes and effects.

The first seven items listed below involved massive outages triggered accidentally by local events, each of which compromised overall system and network survivability. The eighth was triggered by a single human error, but the effects propagated throughout the San Francisco Bay Area. The ninth involved a local outage that was quickly corrected, but whose after-effects continued to propagate for many hours. These cases involved human factors as well as other causes.

- **ARPAnet collapse.** On 27 October 1980, the ARPAnet accidentally shut itself down globally. Collapse, analysis, and recovery took about 4 hours. The problem was due to a hardware design omission (the absence of parity checking in memory), hardware failures (the coexistence of two bogus versions of a node status message resulting from memory errors), and an overly generous algorithm for garbage collection of status messages. Each node in the network became contaminated, memory overflowed, and the network became useless.

- **Internet service outages.** On 23 April 1997, Internet service providers lost contact with nearly all U.S. Internet backbone operators. As a result, much of the Internet was disconnected, some parts for 20 minutes, some for as long as 3 hours. The problem was attributed to MAI Network Services in McLean, Virginia (www.mai.net), which provided Sprint and other backbone providers with incorrect routing tables, the result of which was that MAI was flooded with traffic. In addition, the InterNIC directory incorrectly listed Florida Internet Exchange as the owner of the routing tables. A "technical bug" was also blamed for causing one of MAI's Bay Networks routers not to detect the erroneous data. Furthermore, the routing tables Sprint received were designated as optimal, which gave them higher credibility than otherwise. Something like 50,000 routing addresses all pointed to MAI.

- **Internet domain blockage.** On 16 July 1997, Network Solutions Inc. attempted to run the autogeneration of the top-level domain zone files, which resulted from the failure of a program converting Ingres data into the DNS tables, corrupting the .com and .net domains in the top-level domain name server (DNS), maintained by NSI. Quality-assurance alarms were evidently ignored and the corrupted files were released at 2:30 A.M. EDT on 17 July — with widespread effects. Other servers copied the corrupted files from the NSI version. Corrected files were issued 4 hours later, although various problems lingered.

- **Long-distance calling blockage.** On 15 January 1990, the AT&T long-distance network suffered a nationwide congestion problem that effectively shut down long-distance calling for more than 9 hours. The problem was traced to a flaw in the recovery software in each Signaling System 7 switch that enabled each neighboring switch to crash when receiving traffic from a newly crashed switch that had attempted to reinitialize itself. This crash phenomenon propagated repeatedly throughout the entire network, and resulted in almost no long-distance calls getting through.

- **AT&T frame-relay outage.** On 13 April 1998, AT&T's packet-data frame-relay network collapsed throughout the United States. This network is used for business customers, credit-card activities, bank transactions, travel reservations, among others. The outage resulted from a faulty software upgrade of a circuit card in a single switch, and effectively created a black-hole-like path that shut down the entire network, in some cases for as long as 26 hours.

- **Western power outages.** Electric power outages on 2 July 1996 affected at least 10 Western states. The outage was reportedly triggered by a single tree in Idaho coming in contact with a transmission line, although many other events conspired to create the massive propagation. Further extensive West-Coast outages on 10 August 1996 affected 8 million customers in 8 states, Canada, and Baja (Mexico).

- **Galaxy IV satellite failure.** The loss of the orientation of the Hughes HS601 Galaxy IV satellite on 19 May 1998 (resulting from a malfunction of both the primary system *and* the backup system) caused as many as 40 million pager systems to fail across the United States, as well as the loss of many other services also provided by that satellite — affecting point-of-sale devices, hospital operations, and many other activities. It took several days to reconfiguring the communications using other satellite facilities. (There were two other failures of Hughes HS601 satellites — Galaxy VII on 14 June 1998 and another on 4 July 1998 affecting 3.7 million DirecTV subscribers — but the backup systems worked properly in both of those cases.)

- **San Francisco Bay Area power outage.** At 8:15 A.M. on the morning of 8 December 1998, a power surge resulted from an attempt to reconnect a power station to the grid — but without first having removed a temporary ground connection. Over a million people were affected, some of whom were without power for as long as 8 hours. San Francisco Airport was closed for 1.5 hours. The Pacific Stock Exchange, Rapid Transit, ATMs, offices, and hospitals were without power. There were all sorts of secondary effects; for example, the surge caused SRI only a momentary blip, but that was enough to take many computers down for hours. (RISKS, vol. 20, nos. 11 and 12)

- **Kansas City power outage triggers national air-traffic snarl.** At the Kansas City (Olathe) Air Route Traffic Control Center, at 9:03 A.M. CST on 18 December 1997, a technician routed power through half of the redundant "uninterruptible" power system, preparatory to performing annual preventive maintenance on the other half. Unfortunately, he apparently pulled the wrong circuit board, and took down the remaining half as well. The maintenance procedure also bypassed the standby generators and emergency batteries. The resulting outage took out radio communications with aircraft, radar information, and phone lines to other control centers. Power was out for only 4 minutes, communications were restored shortly thereafter, and backup radar was working by 9:20 A.M. However, at least 300 planes were in the Olathe-controlled airspace at the time, and the effects piled up nationwide. Hundreds of flights were canceled, diverted, or delayed. There were many delays as long as 2 hours, with some delays continuing into the evening.

The remaining cases noted here are examples of other types of accidental survivability problems, although less widespread in their resulting effects.

- **Navy on-board problems.** Two battle cruisers (the USS Hue City and USS Vicksburg) were put out of commission because of difficulties in integrating new onboard weapon-control system software, involving 8 million lines of code (RISKS vol. 19, no. 86). The Navy's Windows NT based Smart Ship technology is also causing potentially serious difficulties. For example, in September 1997, the Aegis missile cruiser USS Yorktown suffered a systems failure during maneuvers off the coast of Cape Charles, Virginia, as the result of an unchecked divide-by-zero in an NT application. The ship was dead in the water for 2 hours and 45 minutes. An earlier loss of propulsion also occurred on 2 May 1997 (RISKS vol. 19, no. 88).

- **Tomahawk missile abort.** On 2 August 1986, a Tomahawk missile suddenly made a soft landing in the middle of an apparently successful launch. The abort sequence was accidentally triggered as a result of a bit dropped by the hardware, possibly due to stray electromagnetic radiation (a cosmic ray hit or other form of interference?) on the computer.[6] (On 8 December 1986, an earlier Tomahawk cruise missile launch crashed because its midcourse program had been accidentally erased on loading.)

---

[6] The cause of this abort is still unknown to me. If you know the results of the final analysis, *please* let me know.

- **Black Hawk helicopter crashes.** In tests, radio waves triggered a complete hydraulic failure of a UH-60 Blackhawk helicopter, effectively generating false electronic commands. Twenty-two people were killed in five Black Hawk crashes before shielding was added to the electronic controls. Failures in other systems have also been linked to electromagnetic interference and rotor control failures. (On 13 December 1998, *60 Minutes* reported that repeated notices of failures had not been acted upon.)

- **Electromagnetic interference on defense systems.** Patriot defenses and Predator unmanned aerial vehicles reportedly cannot work properly in certain foreign countries (Germany, Japan, South Korea, and Bahrain are particular instances) because of frequency clashes. For example, Patriot missile system radios, radars, and data-link terminals clash with Korean cellular phones; pagers of U.S. forces clash with Japanese aeronautical systems; crib monitors used on U.S. bases clash with German telephone service. In Bahrain, SPS-40 and SPS-49 radars are unusable because of interference from the national telecommunications services. (See *Defense Week* issue released 26 October 1998.)

- **Phobos probe losses.** The Phobos I probe was doomed by a faulty software update, which caused a loss of solar orientation, which in turn resulted in discharge of the solar batteries. Phobos II encountered a similar fate when the automatic antenna reorientation failed, causing a permanent loss of communications. Several similar catastrophes are linked to faulty maintenance.

- **Other space cases.** Other cases of nonsurvivable systems are also worth noting: (1) the cosmic ray bombardment of TDRS in 1984 (which cut the Challenger's communications in half), (2) the ill-fated Challenger launch in 1986 due to an O-ring weak-link problem, (3) sunspot activity that affected the computers and altered Skylab's orbit in 1979, and (4) an Atlas-Centaur whose program was altered by lightning.

- **Patriot missile defense.** In addition, many cases have occurred in which system functionality continued but the overall performance was no longer consistent with expectations. An example is provided by the Patriot software whose clock drifted far enough to prevent it from adequately tracking the incoming SCUD that hit the Dhahran barracks. It might be said that the computer hardware survived (it continued to do its computations), but the necessary application functionality did not survive over the 100 hours that the system had been running (instead of the 14 hours specified by the requirements). This illustrates the fact that we must always define survivability with respect to specific requirements or expectations.

- **Cable cuts.** There have been numerous cases in which a single cable was cut, with amazingly dispersed effects — in one case resulting in prolonged delays at all three airports in the New York City area. There have also been cases in which multiple circuits have been severed simultaneously. In 1986, the entire Northeast of the United States was separated from the rest of the ARPAnet because all seven circuits actually went through the same conduit in White Plains. In 1991, two fiber-optic lines in Annandale, Virginia, that had been intentionally placed in separate conduits to avoid such single weak links were simultaneously severed, affecting 80,000 telephone circuits (including the Pentagon and news services).

Next, we consider a few cases attributed to malicious acts.

- **Australian sabotage.** A massive communications blackout in Sydney, Australia, on 22 November 1987 was caused by a knowledgeable saboteur who had been a former Telecom employee. The attacker severed 24 main cables in 10 different locations, which had been carefully selected to have maximum effect. This attack knocked out 35,000 telephone lines, shutting down many computers, banks, telephone offices, ATMs, point-of-sale systems, stores, telexes, facsimile, and betting-office services. Because all international services are routed through Sydney, the effects were not just local. It was suspected that the attack had been based on 2-year-old information, because the same attack 2 years earlier would have been completely devastating.[7]
- **San Francisco blackout attributed to sabotage.** 126,000 customers in northern San Francisco experienced a power outage for as long as 3.5 hours beginning at 6:15 A.M. on 25 October 1997, when five transformers at a single power station stopped working. The FBI counterterrorism unit investigated what it considered to be sabotage, whereby 39 of the 42 switches at one substation appeared to have been manually opened. (See RISKS-19.42.)
- **Other malicious attacks.** Many system outages have resulted from malicious attacks. In the world of conventional computer systems, there have been various malicious attacks using penetrations and Trojan horses, including a variety of logic bombs and time bombs inserted by disgruntled employees, a tampered database that affected General Dynamics' Atlas rocket (resulting in the conviction of the perpetrator – see RISKS-11.95), a surreptitiously changed password that effectively blocked usage of a Washington, DC city computer system (in retaliation for alleged reactions to the saboteur's criticism of his superiors), and the apparently intentional malicious deletion of the annual financial records for a town in Arizona. In many other cases, a system superficially appeared to be operating correctly, but had actually been tampered with or penetrated in such a way that much greater damage could have occurred. In the personal computer world, some of the "viruses" have had serious consequences in disabling systems.

Many cases of pranks and nuisance acts also clearly demonstrate how vulnerable systems are to attack, but that are less serious in nature.

- **Hacked Web sites.** Many organizations now depend on their Web sites for dissemination of timely and reliable information. However, serious risks exist of databases being altered by intruders. Such intrusions continue to occur, because the systems remain vulnerable. Here are a few recent cases, although they were mostly intended as pranks rather than malicious activities:

---

[7] "Saboteur Tried To Black Out Australia" in *The Australian*, 23 November 1987, Sydney, Australia, p.1, 2nd edition, reprinted in ACM *Software Engineering Notes*, vol. 13, no. 1, January 1988, pp. 15-16.

  - ⋆ Justice Department (RISKS-18.35)
  - ⋆ CIA (RISKS-18.49)
  - ⋆ Three Army Web sites (RISKS-19.63)
  - ⋆ Air Force (RISKS-18.64)
  - ⋆ NASA (RISKS-18.88)
  - ⋆ National Collegiate Athletic Association (RISKS-18.88)
  - ⋆ Swedish meatpacker site (RISKS-19.14)

  There was also a report in *Federal Computer Week* that a DoD bloodtype database had been subverted and bloodtype data altered (RISKS-19.97); however, that report was subsequently corrected: apparently there had been no such penetration, although a red team had identified the possibility of such an attack and contemplated its possible effects (RISKS-20.02). The fallacious report apparently did cause the Pentagon to reconsider what information is put on its Web sites.

- **Internet Service Provider and server attacks.** The PANIX ISP suffered a severe denial of service resulting from a `syn flooding` attack (RISKS-18.48). WebCom did also (RISKS-18.69).

- **Faked e-mail.** Given the ease with which e-mail can be spoofed, it is not surprising to find various cases involving phony names, e-mail addresses and IP addresses, bogus content, and even altered text on legitimate messages.

References to these and many other similar cases of nonsurvivable systems and networks can be found in Neumann's *RISKS* book [228] and in the on-line archives of the Risks Forum at `http://catless.ncl.ac.uk/Risks/`, where you can browse the most recent issue of RISKS. A compendium of short, mostly one-liner, descriptions of cases ([233], online at `ftp://ftp.csl.sri.com/pub/users/neumann/illustrative.ps` and `.pdf`). (Other known cases have been reported informally, but not documented publicly.) Some cases of nonsurviving systems are attributable to software flaws that were introduced by system design, by system software development, or by maintenance, at various points in the system life cycle. Some were due to hardware, others to environmental factors such as electromagnetic radiation, others simply to human foibles.

Malicious system misuse is a very serious potential problem whenever it can result in system collapse, although most of the penetration efforts recorded to date were attacks on computer systems themselves rather than on critical applications that used computers. Nevertheless, serious security vulnerabilities exist in many mission-critical systems, many of which could result in loss of survivability.

1.7 Causes and Effects

Breakdowns in system survivability are often attributed to either security problems or reliability problems. However, there is an interesting crossover between the two types of problems, whereby causes and effects may be related and in some cases intermixed. The following enumeration suggests this coupling. It illustrates the distinctions and similarities between the two types, and gives a preliminary view of some of the interdependencies.

- **Reliability problems that also could have been security problems.** In some cases, failures that have been traced to system reliability problems could alternatively have been caused by malicious human actions. For example, both the 27 October 1980 ARPANET collapse and the 15 January 1990 AT&T collapse could have been maliciously triggered, although the system hardware and software flaws that permitted those problems were already in place, inadvertently. In the former case, it would have been possible to maliciously insert bogus status messages into the node traffic, and create exactly the same effect of saturating every node in the network. In the latter case, Neumann was told by two young intruders shortly afterward that they had been experimenting on the first switch when the initial crash occurred; in that case, it is quite conceivable that the acknowledged reliability problem may actually have been triggered as an accidental byproduct of a security penetration — if they had caused the initial crash. Indeed, any distributed system with remote maintenance facilities has simultaneous potentials for unreliability modes and malicious compromises; remote maintenance is very common in many systems, including via dial-up lines and Internet access (in addition to hard-wired connections). Similarly, there have been many inadvertent cases of accidentally severed cables where the same effect could have been achieved intentionally — if one knew where to dig. Once an accidentally exploitable vulnerability becomes known, it may be relatively easy to create similar conditions maliciously.

- **Security problems that could also have been reliability problems.** In certain cases, system disruptions that have been traced to malicious human actions could alternatively have been caused by hardware or software malfunctions. This is especially true with denial-of-service attacks in which an intentionally exploited single weak link could just as well have failed accidentally. It is even true where multiple events could have been triggered by a single underlying cause — as could have been the case in the San Francisco power outage noted above if there had been a previously unidentified common-mode failure mode that accidentally tripped multiple switches.

- **Reliability problems that are less likely to occur as security problems.** Some cases have resulted from an obscure combination of malfunctions or other reliability problems that could not realistically have arisen from malicious human actions. For example, in the ARPAnet collapse, there was a fundamental design decision to omit hardware-implemented parity checking of stored status words (parity checking was present for transmissions), and a weak garbage collection algorithm in software. The result was clearly a network-wide reliability problem that was unlikely to have been created intentionally as a Trojan horse or installed dynamically through a security penetration. There were much easier ways to trash the network. (On the other hand, once this vulnerability had been recognized, it would have been very easy to trigger the same failure mode maliciously, as noted in the first bulleted item above.) Similarly, the Patriot missile clock-drift problem was a subtle implementation flaw that would not likely have been created intentionally — unless the development process itself had been subverted; the vulnerability was fairly subtle, and arose only because of prolonged use of the missile platforms in the same location, without the expected reinitialization. Nevertheless, in such cases of obscure programming bugs, it is possible that a disgruntled programmer could have planted a much simpler undetectable Trojan horse in the software to achieve comparable results.

- **Security problems that are less likely to occur as reliability problems.** Cases that result from extremely elaborate malicious human actions are much less likely to be caused by malfunctions. For example, consider a complicated sequence of successive penetrations and manipulations of programs and data (e.g., cracking, hacking, and Trojan horsing), relying at each step on knowledge gained from the efforts thus far. In such cases, freak accidents are much less probable than intelligently coordinated attacks (as in the case of the Australian sabotage noted above).

In time of crisis, there can be uncertainty over whether a particular survivability problem is related to security or to reliability, availability, and fault tolerance.

- Many systems are expected to fail in unanticipated ways beginning at midnight between 31 December 1999 and 1 January 2000. At that time, a malicious security attack could be very difficult to distinguish from the Y2K problem itself.
- Prevention against denial-of-service attacks is typically considered to be a security problem, whereas design for high system availability is considered to be a reliability problem. Clearly, they are closely interrelated.
- Protecting against electromagnetic interference has attributes of both security and reliability, because it could occur maliciously (jamming) or accidentally (e.g., atmospheric radiation, lightning, circuit emanations). Such interrelationships are considered in Chapter 3.

Furthermore, in certain cases it may not be evident whether a particular attack was natural or human-related — and if human, whether accidental or intentional, malicious or otherwise. Indeed, there is long-standing evidence that intruders ("crackers") have had access to the telephone switches, and could have caused results otherwise attributed to system problems. As noted above, the 15 January 1990 AT&T outage may actually have been triggered by intruders, albeit accidentally. There is also an unverified statement made by an FBI agent during a talk at the University of California at Davis to the effect that the 2 July 1997 West coast power outage involved some maliciously caused events.

As further examples of the fuzzy crossover between reliability and security – although directed more toward survivability of integrity requirements than toward survivability *per se* — there have been numerous cases of suspicious activities involving computers used in elections. In one case in particular, the results of the preliminary test processing were left undeleted, and actually would have caused the wrong winner to be elected, had an anomaly not been detected. Although this error was eventually diagnosed and corrected, the claim was of course made that this was an accident. How do you know it was not intentional?

The foregoing discussion also applies to performance degradations as well as complete outages. The evident heterogeneity of causes and effects suggests that systems should be developed to anticipate a broader class of threats — not just to narrowly address threats to security, or to reliability, or to performance, but rather to address the necessary requirements in the same context.

An obvious conclusion of this discussion is that systems should be designed to be survivable and robust, reliable and secure, to withstand both accidental malfunctions and intentionally caused outages or other deviations from desired behavior.

Chapter 2

Threats to Survivability

Numerous vulnerabilities, threats, and risks are encountered in attempting to develop, operate, and maintain systems with stringent survivability requirements. All these sources of adversity can result in system and application survivability being undermined. The sections of this chapter consider threats to security, reliability, and performance, respectively. Whereas it is convenient to think of these types of threats as independent of one another, they are in fact related in various ways. However, what is most important is that the totality of threats must be addressed by the system requirements and by the system architectures that presume to address those requirements.

2.1 Threats to Security

> *Security is mostly a superstition.*[1]
> Helen Keller

Malicious attacks can take many forms, summarized in Table 2.1 according to a classification scheme shown in Figure 2.1, based on earlier work of Neumann and Parker [239]. For visual simplicity, the figure is approximated as a simple tree. However, it actually represents a system of descriptors rather than a taxonomy in the usual sense, in that a given misuse may involve multiple techniques within several classes.

The order of categorization depicted is roughly from the physical world to the hardware to the software, and from unauthorized use to misuse of authority. The first class includes external misuses that can take place without any access to the computer system. The second class concerns hardware misuse and typically requires some involvement with the computer system itself: two examples in this class are eavesdropping and interference (usually electronic or electromagnetic, but optical and other forms are also possible). The third class includes masquerading in a variety of forms. The fourth includes the establishment of deferred misuse, for example, the creation and enabling of a Trojan horse (as opposed to subsequent misuse that accompanies the actual execution of the Trojan-horse program — which may show up in other classes at a later time), or other forms of pest programs discussed below. The fifth class involves bypass of authorization, possibly enabling a user to appear to be authorized — or not to appear at all (that is, to be invisible to the audit trails). The remaining classes involve active and passive misuse of resources, inaction that might result in misuse, and finally misuse that helps in carrying out additional misuses (such as preparation for an attack on another system or use of a computer in a criminal enterprise).

---

[1] Belief or practice resulting from ignorance, fear of the unknown, or trust in magic or chance (Webster)

The main downward sloping right-hand diagonal line in Figure 2.1 indicates typical steps and modes of intended use of computer systems. The leftward branches all involve misuse, while the rightward branches represent potentially acceptable use — until a leftward branch is taken. (Each labeled mode of usage along the main-diagonal intended-usage line is the antithesis of the corresponding leftward misuse branch.) Every leftward branch represents a class of vulnerabilities that must be defended against — that is, either avoided altogether or else detected and recovered from. The means for prevention, deterrence, avoidance, detection, and recovery typically differ from one branch to the next. (Even inaction may imply misuse, although no abusive act of commission may have occurred.)

The ordering used in Figure 2.1 and Table 2.1 is in some sense upside down from the natural layering used in Tables 3.1 and 1.1 — except for the External Misuse category, which is at the top. This order helps to maintain the sense of the cumulatively increasing binary-tree choices at each layer and the successful choices down the right-sloping diagonal.

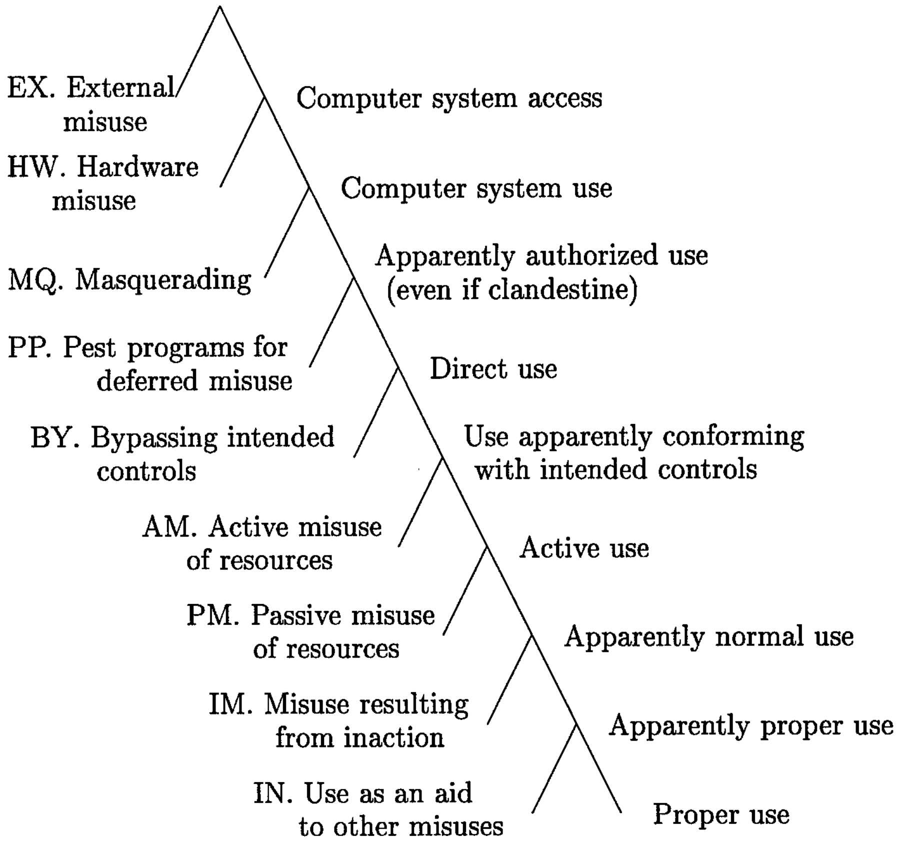
The two security classes of primary interest here are bypasses of authority (trapdoor exploitations and authorization attacks) and preplanned pest programs such as Trojan horses, viruses, and worms, with effects including time bombs, logic bombs, and general havoc. However, several other forms are also important in the present context, and these are also discussed.[2]

2.1.1 Bypasses

- **Bypassing intended controls.** Trapdoors and other system flaws often enable controls to be bypassed. Bypassing may involve circumvention of existing controls, modification of those controls, or improper acquisition of otherwise denied authority, presumably with the intent to subsequently misuse the acquired access rights. Bypassed controls might involve operating-system authentication mechanisms, system or database access controls, and firewalls. In the World Wide Web, attacks may exploit flaws in the programming environment or in browsers and servers. For example, with respect to the Java language, weaknesses have been found in the language, in the virtual machine model (JVM), and in the Java Development Kit; although many of those vulnerabilities have been fixed or reduced [105, 107], some still remain. Similar problems exist within the Microsoft Internet Explorer environment, although the penetrations tend to have greater potential damaging effects than in JDK1.2 (which has finer-grain access controls), because of the lack of an enforceable security policy other than merely digitally signed applets.

---

[2] The following description is adapted from Neumann and Parker [239], and in turn evolved from some earlier classifications discussed by Neumann [223]. An extensive discussion of each misuse class is presented by Neumann and Parker [240], along with examples of each class. They also present an analysis of over two hundred cases [238].


EX. External misuse
Computer system access
HW. Hardware misuse
Computer system use
MQ. Masquerading
Apparently authorized use (even if clandestine)
PP. Pest programs for deferred misuse
Direct use
BY. Bypassing intended controls
Use apparently conforming with intended controls
AM. Active misuse of resources
Active use
PM. Passive misuse of resources
Apparently normal use
IM. Misuse resulting from inaction
Apparently proper use
IN. Use as an aid to other misuses
Proper use

Adapted from P.G. Neumann and D.B. Parker,
"A summary of computer misuse techniques," Twelfth National
Computer Security Conference, Baltimore, Maryland, 1989.

Figure 2.1: Classes of Computer Misuse Techniques

Table 2.1: Types of Computer Misuse

External misuse (EX)

1. Visual spying: observation of keystrokes or screens
2. Misrepresentation: deception of operators and users
3. Physical scavenging: dumpster-diving for printout

Hardware misuse (HW)

4. Logical scavenging: examining discarded or stolen media
5. Eavesdropping: electronic or other data interception
6. Interference: electronic or other jamming
7. Physical attack on, or modification of, equipment or power
8. Physical removal of equipment and storage media

Masquerading (MQ)

9. Impersonation: false identity external to computer systems
10. Piggybacking attacks on communication lines, workstations
11. Playback and spoofing attacks
12. Network weaving to mask physical whereabouts or routing

Pest programs (PP) — setting up opportunities for further misuse

13. Trojan-horse attacks (including letter bombs)
14. Logic bombs (a form of Trojan horse, including time bombs)
15. Malevolent worm attacks, acquiring distributed resources
16. Virus attacks, attaching to programs and replicating

Bypassing authentication or authorization (BY)

17. Trapdoor attacks, from any of a variety of sources:
    a. Improper identification and authentication
    b. Improper initialization or allocation
    c. Improper termination or deallocation
    d. Improper runtime validation
    e. Naming flaws, confusions, and aliases
    f. Improper encapsulation: exposed implementation detail
    g. Asynchronous flaws: e.g., time-of-check to time-of-use anomalies
    h. Other logic errors
18. Authorization attacks, for example, password cracking, token hacking

Active misuse of authority (AM) (writing, using, with apparent authorization)

19. Creation, modification, use, service denials (includes false data entry)
20. Incremental attacks (e.g., salami attacks)
21. Denials of service (including saturation attacks)

Passive misuse of authority (reading, with apparent authorization) (PM)

22. Browsing randomly or searching for particular characteristics
23. Inference and aggregation (especially in databases), traffic analysis
24. Covert channel exploitation and other data leakage

25. **Misuse through inaction** (IM): willful neglect, errors of omission
26. **Use as an indirect aid for subsequent misuse** (IN): off-line preencryptive matching, factoring large numbers, autodialer scanning.

Common cases of unauthorized access can result from system and usage flaws (e.g., trapdoors that permit devious access paths) such as the following [223]:

- Inadequate identification, authentication, and authorization of users, tasks, or systems. Failing to check identity and failing to require nonspoofable authentication are common sources of problems. System spoofing attacks may result in which a system component masquerades as another. The **debug** option of **sendmail** and the **.rhosts** file exploited by the Internet Worm [294, 319, 324] are examples. These exploitations also represent cases in which authentication and authority checking were both absent. (Password-related trapdoors are noted separately below.)
- Improper initialization: improper initial domain selection or security parameters; improper partitioning, as in implicit or hidden sharing of privileged data; embedded operating system parameters in application memory space. If the initial state is not secure, then what follows is in doubt. System configuration, bootload, initialization, and system reinitialization following reconfiguration are all serious sources of vulnerabilities, although often carried out independent of the standard system controls. The **sendmail debug** option exploitation is also a configuration management initialization problem, because system administrators can turn it off. (They tend to leave it enabled for their own convenience.)
- Improper finalization: incompletely handled aborts, accessible vestiges of deallocated resources (residues), incomplete external device disconnects, and other exits that do not properly clean up after themselves. A particularly insidious flaw of this type has existed in several popular operating systems (including TENEX), permitting accidental piggybacking (tailgating) when a user's communication line suffers an interruption; the next user who happens to be assigned the same login port is attached to the still-active user job. This type of problem continues to recur in new systems (although sometimes as a configuration or cabling problem rather than an incomplete termination).
- Incomplete or inconsistent runtime validation: improper argument validation, type checking, and bounds checks; failure to prevent permissions, quotas, and other programmed limits from being exceeded; control bypass or misplacement. One example is the missing bounds check in **gets**, which was exploited by the **finger** attack used in the Internet Worm — enabling the execution of arbitrary code on the overflowed stack buffer. Instances of buffer overflows are very frequent among the archives of bug collectors. (For example, see the BUGTRAQ newsgroup.)
- Improper naming: aliases (e.g., multiple names with inconsistent effects), search-path anomalies, and other context-dependent effects (same name, but with different effects, depending upon the directories, operating system version, and so on).
- Inadequate encapsulation: lack of information hiding, accessibility of internal data structures, alterable audit trails, mid-process control transfers, hidden or undocumented side effects.

- Other sequencing problems: incomplete atomicity or sequentialization, as in faulty handling of interrupts or errors; flawed asynchronous functionality, such as permitting undesired changes to be made between the time of validation and the time of use; incorrect serialization, resulting in harmful nondeterministic behavior, such as critical race conditions[3] and other asynchronous side effects. The so-called time-of-check to time-of-use (ToCtToU) problem illustrates both inadequate atomicity and inadequate encapsulation: an example is the situation in which, after validation is performed on the arguments of a call by reference, the caller can then change the arguments. Also, spoofing attacks and replay attacks can result from predictable or tamperable sequencing — e.g., [203] — or nonmatching sequenced operations – as in SYN attacks.
- Other logic errors: for example, inverted logic, conditions or outcomes not properly matched with branches, wild control transfers (especially if undocumented).

Tailgating may occur accidentally when a user is randomly attached to an improperly deactivated resource, such as a port through which a process is still logged in although its original user is no longer attached. Unintended access may also result from other trapdoor attacks, logical scavenging (e.g., reading a scratch tape before writing upon it), and asynchronous attacks (e.g., incomplete atomic transactions, and discrepancies between time of check and time of use). For example, trapdoors in the implementation of encryption can permit unanticipated access to unencrypted information.

Password attacks are a particularly insidious subclass of trapdoor attacks and may involve, for example

- Guessing of passwords, based on common strings (such as dictionary words and proper names) or strings that might have some logical association with an individual under attack (such as initials, spouse's name, dog's name, and social security number).
- Capture of unencrypted passwords in transit (via local or global net, or by trapdoors such as using /dev/mem for reading the entire Unix memory), whether or not the password file is stored in encrypted form.
- Derivation of passwords by algorithmic means. Various techniques are vastly more effective than exhaustive enumeration. For example, if passwords are known to have a particular form (because they are algorithmically generated, for example), that algorithm can be used to attack them. Passwords can sometimes be derived by inference, as in the TENEX **connect** flaw using a match-and-shift approach to detect page fault activity on the next serially checked password character. Dictionary attacks are particularly popular these days, whereby each entry in a particular on-line dictionary is encrypted, and a match sought with the corresponding entry in the encrypted password file. Such preencryptive dictionary attacks were

---

[3] A race condition is a situation in which the outcome is dependent upon internal timing considerations. A race condition is *critical* if something else depending upon that outcome may be affected by its nondeterminism, and is *noncritical* otherwise.

  described by Bob Morris and Ken Thompson [202] and were exploited in the Internet Worm. Use of foreign-word dictionaries is also common.

  - Existence of unintentional universal passwords, similar to skeleton keys that open a wide range of locks. In at least one case, it was possible to generate would-be passwords satisfying all login routines for a given system even without knowledge of a valid password. In that case, a missing bounds check permitted an intentionally overly long password consisting of a random sequence followed by its encrypted form to overwrite the stored encrypted password and defeat the password checking program [352]).
  - Existence of trapdoors that require no passwords at all, as in the remote login from a similarly named but bogus foreign user (e.g., using the Unix **.rhost** facility). The philosophy of a single universal login authentication for a particular user that is valid throughout all parts of a distributed system and — worse yet — throughout a network of networks represents an enormous vulnerability, because one compromise anywhere can result in extensive damage throughout.
  - Interrupting during authentication (e.g., logging in with an incorrect password, interrupting, and discovering oneself effectively authenticated — which results from inadequate atomicity). This was a problem on several early higher-layer authenticators (that is, not in the operating system).
  - Editing an inadequately protected password file to insert a bogus but viable user identifier and password (improper encapsulation, authentication, and authorization).
  - Inserting a trapdoor into an authorization program within the security perimeter. The classical example is the C-compiler Trojan horse described by Thompson [335] and discussed in Section 1.3.

The variations within this class are amazingly rich. There are also comparable attack techniques for the more sophisticated authentication schemes, such as challenge-response schemes, token authenticators, and other authenticators based on public-key encryption. Even if one-time token authenticators are used for authentication at a remote authentication server, the presence of multiple decentralized servers can permit playback of a captured authenticator within the time window (typically on the order of three 30-second intervals), resulting in almost instantaneous penetrations at the *other* servers. Predictable behavior of message hashing was used to break Netscape's cryptographically based security. In addition, Drew Dean has recently found some intriguing theoretical weaknesses in well-known hashing algorithms (MD4, MD5, SHA-1) used for authentication and integrity checks [82].

2.1.2 Pest Programs

- **Pest programs and setting up subsequent misuses.** Pest program attacks involve the creation, planting, and arming of software by using methods such as Trojan horses, logic bombs, time bombs, letter bombs, malicious worms, and viruses.

A Trojan horse is typically a computer program that surreptitiously contains functionality that contains a hidden source of risk, with effects of varying seriousness. A logic bomb is a Trojan horse whose effects are triggered by the occurrence of some logical event. A time bomb is a logic bomb whose logical trigger is based on time. A virus is a program that can iteratively infect other programs with copies of itself. Personal computer viruses tend to propagate as a result of manual actions such as shared diskettes; mainframe viruses in theory could propagate automatically, but are rare. A worm is a program capable of executing pieces of itself simultaneously, often remotely. These are discussed further below.

Anything that appears innocently as data but whose execution can be triggered surreptitiously represents a serious risk. Web browers present numerous opportunities for the execution of Trojan horses on your own machine when your browser downloads an applet, because the applet typically executes with all or many of your execution privileges. This is a problem with Java applets, ActiveX components, and browser plug-ins. Mobile code can cause some particularly nasty security problems, especially if it originates from an untrustworthy site — where any Web browser may access a site that houses a stable full of Trojan horses or permit other types of intrusions [95, 150, 296], unbeknownst to their users. (For an incisive account of mobile-code security, see Gary McGraw and Ed Felten's Java security book – [187] or its soon-to-appear second edition [188].) Many other situations also have similar risks. Word Macro viruses and the presence of an executable printing-language interpreter (e.g., for PostScript) offer further opportunities for compromise from outside that can set up compromise from within that result from a Trojan horse that would seem to be ordinary data. Similar opportunities arise in CD-ROMs, zip drives, and other portable storage media, agent software, scripting languages, and e-mail enclosures (such as MIME). This problem is likely to worsen in the presence of real-time audio and video, where some enormous security vulnerabilities already exist.

The setting up of these pest programs may actually employ misuses of other classes such as bypasses or misuse of authority, or may be planted via completely normal use, as in a letter bomb. The subsequent execution of the deferred misuses may also rely on further misuse methods. Alternatively, execution may involve the occurrence of some logical event (e.g., a particular date and time, or a logical condition), or may rely on the curiosity, innocence, or normal behavior of the victim. Indeed, because a Trojan horse typically executes with the privileges of its victim(s), its execution may require no further privileges. For example, a Trojan-horse program might find itself authorized to delete all the victim's files. A Trojan-horse letter bomb (with hidden control characters and escape sequences squirreled away in the text) might be harmless unless explicitly read interpretively or otherwise executed; however, if the system permits the transit of such characters, the letter bomb would be able to exploit that flaw and be executed unknowingly by the victim. Several existing systems still permit the interpretation of characters, despite the long-term knowledge of this problem.

2.1.3 Resource Misuse

In addition to the foregoing two forms of malicious attacks (bypasses and pest programs), various forms of attack are related to the misuse of conferred or acquired authority. Indeed, these are the most common forms of attack in some environments:

- **Active misuse of resources.** Active misuse of apparently conferred authority typically alters the state of the system, or changes data (e.g., *data diddling*), or both. Examples include misuse of administrative privileges or superuser privileges; changes in access control parameters to enable other misuses of authority; harmful data alteration and false data entry; denials of service (including saturation, delay, or prolongation of service); aggressive covert channel exploitation, for example, unusual resource use resulting from direct user action or by embedded Trojan horses; and the somewhat exotic salami attacks in which numerous very small pieces (e.g., round-off) are collected (e.g., for personal or corporate gain). The apparently conferred authority may have been obtained surreptitiously, but its use appears locally as if it were legitimate.

- **Passive misuse of resources.** Passive misuse of apparently conferred authority typically results in reading of information without altering any data and without altering the system (with the exception of audit-trail and status data). Passive misuse includes browsing (without specific targets), searching for specific patterns, accessing data aggregates that are more sensitive than the individual items [175], drawing inferences (e.g., as in traffic analysis that permits information to be derived without any access to data), and nonaggressive exploitation of covert channels (storage or timing channels), for example, resulting from unusual reading activity. These events may have no appreciable effect on the objects used or on the state of the system except for the execution of computer instructions and the resulting audit data. They need not involve unauthorized use of services and storage. Certain events that superficially might appear to be passive misuse may in fact result in active misuse — for example, through time-dependent side effects.

2.1.4 Comparison of Attack Modes

Misuse of authority is of considerable concern here because it can be exploited in either the installation or the execution of malicious code, and because it represents a major threat modality. In general, attempts to install and execute malicious code may employ a combination of the methods enumerated above, as well as others external to the computer systems, such as scavenging of discarded materials, visual spying, deception, eavesdropping, theft, hardware tampering, and masquerading attacks — including playback, spoofing, and piggyback attacks; these are discussed by Neumann and Parker [239]. For example, the Wily Hackers [329, 330] exploited trapdoors, masquerading, Trojan horses to capture passwords, and misuse of (acquired) authority. The Internet Worm [294, 319, 324] attacked four different trapdoors, the **debug** option of **sendmail**, **gets** (used in the implementation of **finger**), remote logins exploiting *.rhost* files, and (somewhat gratuitously) a few hundred passwords obtained by selected preencryptive matching attacks. The result was a self-propagating worm with virus-like infection abilities. It is intriguing that no explicit authorization was required for any of those attacks, and thus in reality no authority was exceeded (although clearly the effects produced by the Internet Worm were serious)!

The most basic pest-program problem is the Trojan horse, which contains code that when executed can have malicious effects (or even accidentally devastating effects). The installation of a Trojan horse often employs system vulnerabilities, which permit penetration by either unauthorized or authorized users. Furthermore, when executing, Trojan horses may exploit other vulnerabilities such as trapdoors. In addition, Trojan horses may cause the installation of new trapdoors. Thus, there can be a strong interrelationship between Trojan horses and trapdoors. Time bombs and logic bombs are special cases of Trojan horses. Letter bombs are messages that act as Trojan horses, containing bogus or interpretively executable data.

A *strict-sense virus*, as defined by Cohen [69], is a program that alters other programs to include a copy of itself. Viruses often employ Trojan-horse effects, and the Trojan-horse effects often depend on trapdoors that are either already present or that are created for the occasion. There is a lack of clarity in terminology concerning viruses, with two different sets of usage, one for strict-sense viruses, another for personal-computer viruses. What are called viruses in original usage are usually Trojan horses that are self-propagating without any necessity of human intervention (although people may inadvertently facilitate the spread). What are called viruses in the personal-computer world are usually Trojan horses that are propagated by human action. Personal-computer viruses are rampant, and represent a serious long-term problem (Section 2.1.6). On the other hand, strict-sense viruses (which attach themselves to other programs and propagate without human aid) are a rare phenomenon — none are known to have been perpetrated maliciously, although a few have been created experimentally.

A worm is a program that is distributed into computational segments that can execute remotely. It may be malicious, or may be used constructively — e.g., to provide extensive multiprocessing, as in the case of the early 1980s experiments by Shoch and Hupp at Xerox PARC [320]. The Internet Worm provides a graphic illustration of how vulnerable some systems are to a variety of attacks. It is interesting that, even though some of those vulnerabilities were fixed or reduced, equally horrible vulnerabilities still remain today. (The argument over whether the Internet Worm was a worm or a virus is an example of a "terminology war"; its resolution depends on which set of definitions is used.)

Subtle differences in the types of malicious code are relatively unimportant. Rather than try to make fine distinctions, it is much more appropriate to attempt to defend against the malicious code types systematically, employing a common approach that is capable of addressing the underlying problems. The techniques for an integrated approach to combatting malicious code necessarily cover the entire spectrum, except possibly for certain vulnerabilities that can be completely ruled out — for example, because of operating environment constraints such as if all system access is via hard-wired lines to physically controlled terminals. Thus, generic defenses are more effective in the long term than defenses aimed only at particular attacks. Besides, the attack modes tend to shift with the defenses. For these reasons, it is not surprising that many of the defensive techniques in the system evaluation criteria can be helpful in combatting malicious code and trapdoor attacks (although the criteria at the lower levels do not explicitly prevent such attacks). It is also not surprising that in general the set of techniques necessary for preventing malicious code is very closely related to the techniques necessary for avoiding trapdoors. The weak-link nature of the security problem suggests a close coupling between the two types of attack, and that defense against one type can be helpful in defending against the other type.

Malicious code attacks such as Trojan horses and viruses are not adequately covered by the existing system evaluation criteria. The existence of such code would typically never show up in a system design, except possibly for *accidental* Trojan horses (an exceedingly rare breed). They are addressed primarily implicitly by the criteria and remain a problem even in the most advanced systems (although the threat from external attack can be reduced if those systems are configured and used properly).

Indeed, differences exist among the different types of malicious code problems, but it is the similarities and the overlaps that are most important. Any successful defense must recognize the differences and the similarities, and accommodate both.

Bull, Landwehr, McDermott, and Choi [158] have drafted a taxonomy that classifies program security flaws according to the motive (intentional or inadvertent), the time of introduction (during development, maintenance, or operation), and place of introduction (software or hardware). They subdivide intentional flaws into malicious and nonmalicious, and — continuing on to further substructure — they provide examples for most of these classifications. However, some distinctions are not made. For example, there is no distinction between the existence of a flaw and its exploitation, where the former may be inadvertent and the latter intentional. Presumably, such problems will be addressed in any subsequent versions of their work.

There seem to be serious problems with trying to partition cases into malicious and nonmalicious intents, because of considerable commonalities in the real causes and considerable overlap among the consequences. Also, problems arise in trying to distinguish among human-induced effects and system misbehavior.

2.1.5 Other Attack Methods

In addition to the attack methods noted above, several others are worth discussing here in greater detail, namely, the techniques numbered 1 through 12 in Table 2.1.

- **External misuse.** Generally nontechnological and unobserved, external misuse is physically removed from computer and communication facilities. It has no directly observable effects on the systems and is usually undetectable by the computer security systems; however, it may lead to subsequent technological attacks. Examples include visual spying (e.g., remote observation of typed keystrokes or screen images), physical scavenging (e.g., collection of waste paper or other externally accessible computer media such as discards — so-called *dumpster diving*), and various forms of deception (e.g., misrepresentation of oneself or of reality), external to the computer systems and telecommunications. Surprisingly, many incidents have involved the disposition of computer systems whose sensitive contents have not been sanitized or deleted.

- **Hardware misuse.** Hardware misuse takes on two basic forms, passive or active.

  - Passive hardware misuse tends to have no immediate effects on hardware or software behavior, and includes logical scavenging (such as examination of discarded computer media) and electronic or other types of eavesdropping that intercepts signals, often without the victims' knowledge. Eavesdropping may be carried out remotely (e.g., by picking up emanations) or locally (e.g., by planting a spy-tap device in a terminal, workstation, mainframe, or other hardware-software subsystem).
  - Active hardware misuse typically has side effects; it includes theft of computing equipment and physical storage media; hardware modifications such as internally planted Trojan-horse hardware devices; interference (electromagnetic, optical, or other); physical attacks on equipment and media, such as interruption of or tampering with power supplies or cooling. These activities have direct effects on the computer systems (e.g., internal state changes and/or denials of service).

- **Masquerading.** Masquerading attacks include impersonation of the identity of some other individual or computer subject; spoofing attacks that take advantage of successful impersonation; piggybacking attacks that gain physical access to communication lines or workstations; playback attacks that merely repeat captured or fabricated communications; and network weaving that masks physical whereabouts and routing via both telephone and computer networks, as practiced by the Wily Hackers [329, 330]. These activities may be indistinguishable from legitimate activity.

The remaining two forms of attack listed in Table 2.1 are somewhat more obscure than those noted above. The penultimate case involves misuse through inaction, in which a user, operator, administrator, maintenance person, or perhaps surrogate fails to take an action, either intentionally or accidentally. Such cases may logically be considered as degenerate cases of misuse, but are listed separately because they may have quite different origins.

The final case in Table 2.1 involves system use as an indirect aid in carrying out subsequent actions. Familiar examples include performing a dictionary attack on an encrypted password file, attempting to identify dictionary words used as passwords, and possibly using a separate machine to make detection of this activity harder ([202]); factoring of very large numbers, attempting to break a public-key encryption mechanism such as the Rivest-Shamir-Adleman (RSA) algorithm that depends upon a product of two large primes being difficult to factor; and scanning successive phone numbers, attempting to identify modems that might be attacked subsequently.

2.1.6 Personal-Computer Viruses

Personal-computer viruses may attack in a variety of ways, including corruption of the boot sector, hard disk partition tables, or main memory. They may alter or lock up files, crash the system, and cause delays and other denials of service. These viruses take advantage of the fact that there is no significant security or system integrity in the system software. In practice, personal-computer virus infection is frequently caused by contaminated diagnostic programs.

The number of distinct personal-computer virus strains has grown from five at the beginning of 1988 to over a thousand early in 1992, and has continued to grow steadily since then. By 1998 numbers exceeding 10,000 are commonly quoted. Many different types of viruses and variant forms exist. The growth in the virus 'industry' is enormous. In addition, we are beginning to observe stealth viruses that can conceal their existence in a variety of ways and distribute themselves. Particularly dangerous is the emergence of polymorphic viruses, which can mutate over time and become increasingly difficult to detect. Ultimately the antiviral tools are limited by their inherent incompleteness and by the ridiculously simplistic attitude toward security found in personal-computer operating systems. Serious efforts to develop survivable systems would do well to avoid today's personal-computer operating systems, although the hardware is not intrinsically bad.

2.2 Threats to Reliability

Threats to system and network reliability can take many forms. They can arise during requirements definition, system specification, implementation, operation, and maintenance. They can originate from hardware malfunctions, operating-system software flaws, network software flaws, application software problems, operational errors (e.g., in system configuration, management, and maintenance), environmental anomalies, and — not to be ignored — human mistakes. Some illustrative types of reliability threats are summarized in Table 2.2.

Essentially every one of the types of threats summarized can represent a fundamental threat to overall survivability. Environmental threats can be particularly devastating, especially if equipment and media are seriously damaged. Loss of power and telecommunications are especially critical, particularly if they last for long periods of time and if alternatives are not readily available. Threats to software and hardware reliability can have pervasive effects, although in some cases they may be surmounted.

2.3 Threats to Performance

System and network performance can be threatened as a result of many of the threats to reliability and security discussed in Sections 2.2 and 2.1, respectively. In addition to those threats, performance threats exist that do not directly stem from reliability or security. Inadvertent saturation of resources is one major class, perhaps because of runaway programs or inadequate garbage collection. Table 3.1 notes some of the concepts on which performance may depend.

2.4 Perspective on Threats to Survivability

Threats to survivability and its subtended requirements exist pervasively throughout all system application areas; throughout the layers of abstraction related to hardware, software, and people (as discussed in this report; see Section 3.3); and throughout the stages of development and use noted in Section 1.5. In particular, threats are pervasive throughout the application

Table 2.2: Illustrative Reliability Threats

Outside-environmental threats
- Environmental problems (earthquakes, floods, etc.)
- Power utility disturbances
- Electromagnetic and other external interference
- Inappropriate user behavior, unavailability of key persons

National-infrastructure threats
- Glitches in telecommunications, air-traffic control, power distribution, and other infrastructures dependent on computer-communication infrastructures

Middleware and application-code threats
- Windows environments: cache management, crashes
- Browser and Web server flaws
- Accidentally corrupted code

Database-specific threats
- DBMS software flaws
- Internal database synchronization and cache management
- Distributed database consistency
- Improper DBMS software upgrades and maintenance
- Improper database entries and updates

Network threats
- Faulty network components (hosts, routers, firewalls, etc.)
- Distributed system synchronization
- Traffic blockage and congestion

Operating-system threats
- OS software design and implementation flaws
- Improper OS configuration
- Improper OS upgrades and maintenance
- Failures of backup and retrieval mechanisms

Software-development problems
- Faulty system design and implementation,
- Poor use of software engineering techniques
- Bad programming practice

Programming-language threats
- Compiler language inadequacies
- Compiler design and implementation flaws

Hardware threats
- Flaws in hardware design and implementation
- Undesirable internal hardware state alterations
- Improper hardware maintenance

Inside-environmental threats
- Internal power disturbances
- Self-generated or other internal interference

systems in the critical national infrastructures as well as computer-communication infrastructures. These threats provide the motivation for the survivability requirements discussed in Chapter 3.

To give a detailed example of the breadth of threats in just one critical-infrastructure sector, consider the safety-related issues in the national airspace, and the subtended issues of security and reliability. (See for example, Neumann's position statement for the International Conference on Aviation Safety and Security in the 21st Century [231].) Alexander D. Blumenstiel at the Department of Transportation in Cambridge, Massachusetts, has conducted a remarkable set of studies [43, 45, 44, 55, 46, 47, 48, 54, 50, 51, 52, 56, 53] over the past 14 years. In his series of reports, Blumenstiel has analyzed many issues related to system survivability in the national airspace, with special emphasis on computer-communication security and reliability.

Blumenstiel's early reports (1985–1986) considered the susceptibility of the Advanced Automation System to electronic attack and the electronic security of NAS Plan and other FAA ADP systems. Subsequent reports have continued this study, addressing accreditation (1990, 1991, 1992), certification (1992), air-to-ground communications (1993), air-traffic-control security (1993), and communications, navigation, and surveillance (1994), for example. To our knowledge, this is the most comprehensive set of threat analyses outside of the military establishment,[4] and the breadth and depth of the work deserves careful emulation in other sectors.

To be more specific, Blumenstiel's early reports included a 1986 assessment [55] of vulnerabilities of the Advanced Automation System (AAS) to computer attacks. The AAS was planned at the time as the next-generation system of air-traffic-control computers and controller displays for installation in all air-traffic-control centers. Blumenstiel's study found vulnerabilities to a range of computer attacks in this system and recommended countermeasures. (In 1999, the FAA is finally beginning to upgrade the displays, replacing technology from the mid-1960s.) The FAA specified the countermeasures as a requirement for this system. Blumenstiel also assessed vulnerabilities of the FAA's National Airspace System Data Interchange Network (NADIN), a packet-switched network for interfacility communication of air-traffic-control data [54]. Based on this assessment, Blumenstiel prepared a security management plan for NADIN that has been implemented in the system to protect critical data transmissions. Another study assessed vulnerabilities of the Voice Switching and Control System (VSCS). The VSCS is a computer that controls the switching of air-traffic-control communications (between controllers and flight crews and between controllers on the ground) at all air-traffic-control centers. Another study [48] identified and assessed risks to air traffic from electronic attacks on the entire National Air Space System, including air-traffic-control computers, radars, switching systems, and automated maintenance information. This study prioritized all the systems in terms of vulnerabilities and the potential impact of successful attacks on air traffic, including the potential for crashes and the cost of potential delays, and estimated the overall risk. Blumenstiel also produced the security plans for FAA systems required by Public Law 100-235, authored FAA's requirements for computer security accreditation (and designed and developed software to automate the accreditation reporting process) and sensitive application certification [50, 51, 52]. He authored the NIST Guidelines [49] on FAA AIS security accreditation. He was the principal author of the June 1993 Report to Congress on Air Traffic Control Data and Communications Vulnerabilities and Security [56]. Additional studies under Blumenstiel's direction involved assessments of air-traffic-control telecommunications systems to electronic attacks, and development of the strategic plan to protect such systems.

---

[4] For further information, contact Alex Blumenstiel at 1-617-494-2391 (Blumenstie@volpe1.dot.gov) or Darryl Robbins, FAA Office of Civil Aviation Security Operations, Internal and AIS Branch.

With respect to the national airspace, and with respect to the other national infrastructures and the computer-communication infrastructures, it is clear that the threats are pervasive, encompassing both intentional and accidental causes. However, it is certainly unpopular to discuss these threats openly, and thus they tend to be largely downplayed — if not almost completely ignored.

In general, it is very difficult for an organization to expend resources on events that have not happened or that are perceived to be very unlikely to occur. The importance of realistic threat and risk analyses is that it becomes much easier to justify the effort and expenditures if a clear demonstration of the risks can be made.

Chapter 3

Requirements and Their Interdependence

Things derive their being and nature by mutual dependence and are nothing in themselves.
Nagarjuna, second-century Buddhist philosopher

We next elaborate on the requirements, threats, risks, and recommendations outlined in [25] and discussed in Chapter 1 of this report, in such a way that those requirements could apply broadly to a wide range of survivable system developments and to the procurement of systems with critical requirements for survivability.

Our approach is rooted in the establishment of a sound basic set of generic requirements for survivability and the explicit determination of how survivability in turn requires other secondary properties. Secondary properties include various aspects of security in preventing willful misuse; reliability, fault tolerance, and resource availability despite accidental failures (with real-time availability when required); certain aspects of functional correctness; ease of use; reconfigurability under duress; and some sense of overall system robustness when faults exceed tolerability. In turn, security requirements include integrity of systems and networking, confidentiality to avoid dissemination of information that could be useful to attackers (especially cryptographic keys and authentication parameters), high availability and prevention of denials of service despite malicious actions, authorization, accountability, rapid detectability of adverse situations, and prevention of other forms of misuse. Reliability requirements include fault tolerance, fault detection and recovery, and responses to unexpected failure modes. Security and reliability have some requirements that are related, such as resistance to electromagnetic and other interference. Furthermore, some of the requirements interact with other requirements, and must be harmonized to ensure that they are not contradictory. Each requirement has manifestations at each layer of abstraction, and corresponding special issues that must be accommodated. Particular layers of abstraction must address relevant properties — of applications, databases, systems, subsystems, and networking software. Types of adversities to be covered by the requirements must include the full spectrum of applicable potential threats, such as malicious software and hardware attacks, system malfunctions, and electronic interference noted above. All reasonable risks must be anticipated and protected against. Thus, our approach is developing a somewhat canonical requirements framework for survivability that encompasses all the relevant issues, and that demonstrates how the different requirements interrelate.

Of particular importance are the ways in which some of these requirements interact with one another, and how when systems are developed to satisfy those requirements, the components supposedly addressing different requirements actually interact with one another. Ideally, it is helpful if those interactions are understood ahead of time rather than manifesting themselves much later in the development process, or in system use, as seemingly obscure vulnerabilities, flaws, risks, and in some cases catastrophes.

For any given application, the specific requirements must be derived from knowledge of the operational environment, the perceived threats, the evaluated risks, many other practical matters such as the expected difficulty and costs necessary to implement those requirements, the available resources in funding and manpower, and considerations of how the peculiarities of the given application are likely to compound the difficulties in development. Mapping the generic requirements onto the detailed specific requirements then remains a vital challenge that must be undertaken before any serious development effort is begun.

3.1 Survivability and its Subrequirements

In defining survivability and some of the requirements on which it most depends — security, reliability, and performance — we work primarily from the perspective of requirements that can be dependably enforced and applied to enhance the overall system and network survivability.

As observed at the beginning of this chapter, numerous properties are necessary for overall survivability, some of which are — or in some cases merely seem to be — interdependent. A highly oversimplified but nonetheless illustrative summary of a portion of the subtended requirements hierarchy is given in Figure 3.1.[1] We have somewhat arbitrarily taken security, reliability, and performance as three major requirements that can contribute in rather basic ways to achieving high-level survivability requirements. (Other conceptual arrangements of the hierarchy are also possible. In addition, it is worth noting that the overarching requirement for human safety is evidently at an even higher level than survivability, because human safety typically depends on overall system survivability and other mission-critical properties.)

In Section 1.2.2 we observe that there are multiple but closely related manifestations of availability. This is depicted graphically in Figure 3.1 by a common node ("Avail") subtended from both security and reliability, and by a comparable node subtended from performance. (The asterisked nodes denote a reconvergence of a common set of requirements across the three major requirements.) Although it is usually desirable to keep these three manifestations of availability separate during a requirements specification and analysis, it is highly advantageous to consider them in an integrated way during system design and implementation. Thus, the specification of the requirements can benefit from an understanding of the ways in which the different manifestations interact with or depend on one another.

---

[1] This is a primitive rendering of the figure, which will be displayed more elegantly later.

```
                     Survivability    [An overarching requirement:
                          /|\         a collection of
                         / | \        emergent properties]
                        /  |  \
                       /   |   \
                      /    |    \
                     /     |     \
                    /      |      \
                   /       |       \
                  /        |        \
                 /         |         \
                /          |          \
           Security  Reliability  Performance   [Major subrequirements]
              /|\         /|\         /|\
             / | \       / | \       / | \
            /  |  \     /  |  \     /  |  \
           /   |   \   /   |   \   /   |   \      [Subtended
       Inte- Conf- Avail  FT Fail  RT  NRT Avail  requirements:
      grity id'ity   *    |\ modes /\  /|\   *      FT=fault tolerance
        /|   |\      |\   | \   /|  \/     /|\     RT=real-time
       / |   | \     | \  |  \ / | Prior-  / | \    NRT=non-real-time]
      /  #   |  \    |  \     #    ities  /
    MLI No MLS Dis- MLA  \    No         /        [More detailed
    / change | cret- |    \ change      /         requirements]
   /    /|   | ion-  |     \           /
  /    / |   | ary   |      * Unified *
 /    /  |   |   |   |      availability
X  Sys Data  X       X      requirements
    /|   |\                       [X = Shared components of MLX!!]
   / |   | \                      [* = Reconvergence of availability]
  /  |   |  \                     [# = Reconvergence of data integrity]
```

Figure 3.1: Illustrative Subset of Requirements Hierarchy

Furthermore, techniques that contribute to more than one of the major requirements can be implemented more uniformly.

3.1.1 Generalized Survivability

As noted at the beginning of Chapter 1, the term *survivability* denotes the ability of a computer-communication system-based application to continue satisfying certain critical requirements (e.g., requirements for security, reliability, real-time responsiveness, and correctness) in the face of adverse conditions. The scope of adversity may in some cases be precisely defined, but more typically it is not well defined.

Some of the adversities can be foreseen as likely to occur, with consequences that can be perceived as potentially harmful; these can be enumerated and defined. Other adversities may be foreseen as unlikely, or as not having serious consequences to worry about — or may not even be anticipated at all. Ideally, appropriate survivability-preserving actions should be taken irrespective of whether the adversity was foreseen or not. Defensive system design and defensive programming practices are necessary to cover otherwise unanticipated events. Although there can clearly be circumstances in which system survival is not possible — for example, when all communication lines and all power are out — some reasonable contingency plans should be in place, even if in the last resort it is merely emergency actions by the operations staff. In addition, the blame in cases of complete outages may rest with insufficient foresight in system design.

Survivability in this sense can be defined only in terms of specific requirements that must be met under incompletely specified circumstances (where those circumstances may possibly be dynamically changing), which will often differ from one type of adversity to another.

Specific survivability requirements typically vary considerably from one application to another. For example, one set of system requirements might allow degraded performance or other real-time dynamic tradeoffs in times of extreme need (e.g., being able to relax certain security requirements in favor of maintaining real-time requirements when under attack).[2] Another set might prioritize the computational tasks and permit degradation according to the established priorities. In general, a system's survivability requirements might specify that the system must withstand attacks (providing integrity and availability aspects of security) and be resistant to hardware malfunctions (providing reliability and hardware fault tolerance), software outages (providing resistance to hardware- or software-induced software crashes), and acts of God (e.g., anticipating the consequences of communications interference, floods, earthquakes, lightning strikes, and power failures) that might otherwise render the system completely or partially inoperative. In this context, survivability appears in the guise of a high-layer system integrity requirement.

Thus, we adopt the following tentative working definitions:

- **Information survivability** relates to the extent to which suitably correct and up-to-date information can be available whenever it is needed. Information survivability is a meaningful concept at different layers of abstraction, including systems, networks, and applications.

---

[2] This approach was taken in the Secure Alpha effort [111], an SRI project that redesigned a real-time system (Jensen's ARCHONS project [133]) to include multilevel security and to permit dynamic tradeoffs to ensure performance.

- **Computer-system survivability** relates to the extent to which a computer system's ability can continue to satisfy certain stated requirements in the face of arbitrary adversities.
- **Network survivability** relates to the extent to which a computer network's ability can continue to satisfy certain stated requirements in the face of arbitrary adversities.
- **Application survivability** relates to the extent to which an entire system application attains system survivability.
- **Enterprise survivability** relates to the extent to which an overall enterprise (such as a business or critical infrastructure) can continue to satisfy certain stated requirements in the face of arbitrary adversities.

System survivability can be defined in terms of an overall application or in terms of specific computer-communication systems, subsystems, or networks. Each type of potential adversity may have its own measure of survivability.

In the definitions above, the term "arbitrary adversities" implies more than merely the ability to withstand "known adversities" or "specified adversities" — it also implies a characterization of the ability to withstand adversities that were not anticipated such as those that exceed the reliability and security tolerances supposedly covered by the design. In each case, a meaningful assessment of survivability rests not only on what happens when an anticipated adversity occurs, but also on what might happen in response to unanticipated events. This requires some determination of the actual coverage, not just the designed coverage with respect to anticipated faults and threats.

Continued enforcement of system integrity, system availability, data confidentiality, and data integrity (for example) are typically fundamental aspects of survivability. Whenever specific lower-layer survivability properties are explicitly included among their constituent system security properties, then survivability can also be considered as a security property (and, specifically, an integrity property). However, for present purposes we consider application survivability as an overarching property to be maintained by the application in its entirety.

3.1.2 Security

Intuitively, the natural-language meaning of security implies protection against undesirable events. System security and data security are two types of security. The three most commonly identified properties relating to security are confidentiality, integrity, and availability. There are also other important forms of security, such as the detection of misuse and the prevention of general misuse that does not necessarily violate confidentiality, integrity, or availability, particularly when committed by authorized users.

With respect to any particular functional layer, the primary attributes of security are summarized as follows:

- **Confidentiality.** Confidentiality involves protection against undesired release of information by the releaser, and protection against undesired acquisition of information by the acquirer. Confidentiality is meaningful with respect to data, but also with respect to the system itself, for example, maintaining the confidentiality of software or of the hardware implementation.

- **Integrity.** Integrity implies remaining in a sound, unimpaired, or otherwise desirable condition. Integrity is meaningful in many ways; it may have somewhat different meanings for a system, a subsystem, an application, data, hardware, communications links, and other entities.

- **Availability.** Availability implies that certain required resources are available when and as needed. Availability can be applied at many levels of abstraction, including systems, subsystems, data entities, and communications links. Prevention of denial of service is an availability requirement, although it also has a system-integrity component.

- **Authentication.** Authentication encompasses determination that a presumed identity (of a user, system, subsystem, or other computational abstraction) is actually valid. It has many forms, such as authenticating a user by a system, authenticating a system by another system, authenticating a system by a user (e.g., to ensure the user is not being spoofed by a bogus system).

- **Trusted paths.** In conventional systems, a user has no real assurance that he or she is actually communicating with the desired system (rather than a masquerader or Trojan-horsed system), and one system has little assurance that it is actually communicating with a second system of its choice (rather than a masquerader or accidental alternative). A communication path whose terminations can be assured without compromise is known as a *trusted path* (although as is often the case in Orange Book parlance of trust, what is really meant is a *trustworthy path*). Trusted paths must be persistent as well as initially authenticated. Other risks include covert channels and timing attacks related to trusted paths (e.g., see [337]).

- **Prevention of misuse.** Above and beyond explicit violations of confidentiality, integrity, availability, and authentication policies, there can be misuses that are undesirable but that do not explicitly violate those policy elements — such as covert channels and small-scale thefts of services, some of which may be too small to be detected. Prevention of misuse encompasses those.

- **Tamperproofing and anti-reverse-engineering techniques.** Tamperproofing is an aspect of integrity that is particularly critical with respect to the security mechanisms themselves, and especially for cryptographic implementations. Mechanisms that seriously impede reverse engineering can also be important to security, in that there is always an element of security by obscurity — particularly in software-implemented cryptography. Techniques for code and data obfuscation can be valuable for making it difficult to reverse-engineer object code and to locate critical data elements such as cryptographic keying information.

- **Auditability and detection of misuse.** For the above-mentioned types of misuse, and particularly for misuse by apparently authorized users (who may actually be either legitimate users or masqueraders), efforts must be made to detect such misuse.

Identification and authentication are essential to the enforcement of confidentiality, integrity, availability, and prevention of generalized misuse, as well as to meaningful misuse detection. They may be either explicitly designated as system security requirements or else subjugated to the implementation, but are fundamental in either case.

Mandatory policies for confidentiality (e.g., multilevel security), integrity, availability, and survivability (MLS, MLI, MLA, and MLX, as introduced in Section 1.2) have the advantage that they cannot be violated by user actions (assuming that the mechanisms are correctly implemented), but have the disadvantage that they may be inflexible for certain kinds of applications. On the other hand, that inflexibility is precisely what makes them powerful organizing architectural concepts.

Covert channels (out-of-band signaling paths) represent potential losses of confidentiality. They are a problem primarily in multilevel-secure systems, in which it may be possible to signal information through inference channels to lower levels, in violation of the security policy. However, covert channels are often not explicitly addressed by system security policies, and are typically not prevented by conventional security access controls: they bypass conventional controls altogether rather than violating them. Avoidance of covert channels is a problem that must be addressed during system design, implementation, and operation. Detection of covert channel exploitation is a problem that must be addressed during operation (see Proctor and Neumann [280]). Two types of covert channels are recognized: storage channels (which exploit the sharing of resources across multiple security levels using normal system functions in unusual ways), and timing channels (which exploit the time-sensitive behavior of a system, perhaps by observing the real-time behavior of a scheduler). Because most existing systems still have overtly exploitable security flaws, covert channels are often of less interest. However, in highly critical applications, they could be an important source of system compromise, for example, with the aid of a Trojan horse that is modulating the covert channel.[3]

Covert channels may also exist with respect to MLI, MLA, and MLX, but they seem less easy to identify.

At each layer of abstraction, each of these concepts may have its own interpretation, in terms of the abstractions at that layer.

Threats to security are considered in Section 2.1.

3.1.3 Reliability and Fault Tolerance

Many requirements for reliability and fault tolerance are appropriate in addressing the various types of threats to reliability summarized in Section 2.2 and Table 2.2. Most of these reliability requirements can have major implications on system and network survivability, in hardware, system software, network software, and application software. In the absence of serious efforts at generalized dependence, the failure of a component may typically result in the failure of higher-layer components that depend on the failed component.

---

[3] Although implications of covert channels in multilevel-secure systems have been downplayed in this report, system and network architectures that militantly seek to avoid covert channels (e.g., see Proctor and Neumann [280]) may have significant benefits for other requirements as well, because the partioning required can contribute to increased survivability.

Some of the reliability concepts are closely tied together with security concepts. For example, reliably high availability with respect to systems and networks is closely related to the prevention of denials of service. Also, degraded performance modes are closely linked with fault tolerance and responses to detected anomalies.

In an early (D)ARPA study, 1972–1973 [236], we recommended that fault tolerance can most effectively be used at each hierarchical layer according to the particular needs of each of the specific abstractions at that layer. That approach is still valid today, and is embodied in the architectural directions pursued in this report. A recent article by Nitin Vaidya [341] further pursues a design principle of multilevel recovery schemes in which the most common cases are disposed of most quickly. (Vaidya considers only two levels, but the concept is readily generalized to more levels, or to a continuous spectrum.)

Numerous examples of survivability failures related to inadequate reliability and fault tolerance are given in Neumann's RISKS book [228], along with a summary of techniques for improving reliability and fault tolerance.

3.1.4 Performance

Complete system and network outages are an extreme form of performance degradation, whether caused accidentally or intentionally, through reliability problems or security problems. However, in some cases even relatively small performance degradations can cause unacceptable behavior, particularly in tightly constrained real-time systems. Thus, performance requirements must be closely coupled with those for security and reliability.

Performance depends on availability in both its security manifestations (e.g., prevention of denials of service) and its reliability manifestations (e.g., fault tolerance and alternative computation modes). This confluence of subrequirements is illustrated in Figure 3.1 as the reconvergence of what is otherwise depicted as a pure tree structure.

The reconvergent nodes indicated by an asterisk (*) in the figure could of course be split into separate but essentially identical nodes if the sanctity of the pure tree structure were important. However, it is not important — and in fact illustrates an important point: apparently different subrequirements that originate from seemingly disjoint requirements are in fact best handled by a common integrated mechanism, rather than treated completely separately. For implementing assured availability, it is true that different techniques may indeed be useful for (1) preventing malicious denials of service, (2) preventing accidental denials of service, (3) preventing failures due to faults that exceed the coverage of fault tolerance, (4) ensuring adequate performance despite intentional acts, (5) ensuring adequate performance despite unintentional acts and system malfunctions, and (6) ensuring adequate performance despite acts of God. On the other hand, by taking a systematic view of these supposedly different aspects of availability, it is likely that many common mechanisms can work synergistically.

Many other interdependencies also exist. For example, the aspect of integrity relating to prevention of undesired changes (to data, programs, firmware, hardware, communications media, and so on) is fundamental to security, reliability, performance, and of course survivability in the large. Several manifestations of the no-unintended-change requirement are indicated by a sharp (#) (the erstwhile octothorpe in early telephony) in the figure, and discussed further in Section 5.11. Similarly, the confidentiality of sensitive information such as cryptographic keys can undermine many desired system properties.

Dependencies in requirements are sometimes not recognized until well into design and implementation. For example, extensive requirements for reliability and availability may induce additional risks with respect to security and survivability, such as those that result from replication of common mechanisms that introduce multiple common vulnerabilities, or the use of different mechanisms that introduce different vulnerabilities. Similarly, extensive security mechanisms may have deleterious effects on performance and system usability. To avoid performance degradation, security controls are often disabled.

3.2 System Requirements for Survivability

As noted, enterprise survivability is a requirement on the enterprise as a whole. Other such highest-layer application requirements might include preservation of human safety for friendly humans, destruction of unfriendly humans by a tactical system in a hostile environment, and detailed accountability of system and human actions in terms of the application functionality.

System survivability typically depends on two types of properties, sometimes called *liveness* properties (implying availability) and *functional safety* properties (implying functional correctness). Alpern and Schneider [8] have shown that every property can be expressed as a combination of functional safety properties and liveness properties, relative to definitions that have evolved from Lamport [152]. Intuitively, functional safety properties imply that nothing bad happens, while liveness implies that eventually something good happens. Indeed, most application-layer requirements (e.g., survivability and human safety) and system-layer requirements have components of each type.

We do not need to make a precise distinction here between the two types of properties. However, we do recognize that some of the desired properties have time-dependent aspects — particularly in highly distributed systems.

Failure of the system or subsystem to enforce any of a variety of properties can result in a loss of application survivability. Some of those necessary underlying properties on which application survivability may depend are illustrated next. In each case, the term *system* can equally well imply an entire computer-communication system or a subsystem thereof.

Some of the necessary properties are largely time independent, although some of them have certain time-dependent attributes. We consider the general properties first and then reconsider those with specific real-time attributes. For simplicity, we include networking and communications issues as an integral part of the system issues, particularly in distributed systems.

Necessary system security properties:

- **System integrity**. Preventing malicious (and, to some extent, accidental) effects on the hardware, system software, and intercommunications.

- **System availability**. Preventing system and communication outages, and even temporary unavailability of resources. Such outages may include malicious or accidental denials of system service.

- **System confidentiality**. Preventing the undesired dissemination or acquisition of sensitive system code and data, particularly if the application can be compromised. Otherwise, for example, knowledge of the system design, a specific algorithm, a piece of code, a password, a cryptographic key, a network authenticator, or a piece of equipment could lead to a system subversion.

- **Authorization and accountability of systems and users**. Assuring that a system is capable of controlling which subsystems and which users are using it. Otherwise, it may be vulnerable to spoofing attacks, penetrations, and other forms of misuse. After any such attack, the system's inability to provide real-time (or at least rapid) accountability and audit-trail analysis may lead to additional compromises of survivability.

- **Data integrity**. Preventing undesired alteration of input data, internal stored data, or output data. Data integrity includes internal data consistency (particularly important in a highly dispersed environment) as well as external consistency with the real world.

- **Data availability**. Preventing disruptions in timely access to data, including sensor data in a control system. Multiple versions of critical data and alternative sensors can help increase data availability.

- **Data confidentiality**. Preventing undesired data disclosure. For example, a penetrator could obtain sensitive data that would compromise the application's ability to fulfill its requirements in a hostile environment.

Necessary network security properties:

- **Network integrity**. Preventing attacks on network nodes and on the communication media that alter network states.

- **Network availability**. Detecting, preventing, and recovering from denial-of-service attacks, such as outages of network nodes and access devices, electromagnetic interference on the communications media. Alternate routes, redundant nodes, and other techniques for fault tolerance can contribute to security as well.

- **Network data confidentiality**. Preventing the undesired interception of unencrypted or decryptable information, typically through the use of encryption, steganography, operations security, and other techniques for defeating acquisition of data and traffic analysis that could be used to affect network and system survivability.

- **Authorization and accountability of network access and control**. All network nodes and access points must be adequately controlled, to reduce the opportunities for spoofing attacks, maintenance attacks, and other forms of misuse. Digital signatures and other cryptographic techniques for authentication can be helpful.

- **Network data integrity.** Preventing undesired alteration of communications themselves — for example, through a combination of error-correcting codes and encryption. This might be considered a system requirement if end-to-end integrity is desired, but can also be a network requirement when link-by-link (node-to-node) protection is needed.

Although the foregoing system and network security issues all imply attempts to constrain usage by users, operators, system programmers, and administrators, they inevitably depend to some extent on compliant system use by those people. Misuse by apparently authorized individuals is a serious potential problem in many applications.

Necessary system and network reliability properties:

- **Fault tolerance.** Preventing undesirable effects resulting from failures of underlying hardware components, subsystems, or indeed the entire system. Essentially, fault tolerance is both a system integrity issue and a system reliability issue. Constructive use of redundancy is essential. Survivability is a particular concern when the nominal fault-tolerance coverage is exceeded.

- **Functional correctness.** Assuring that a flaw in the application or in the computer operating system, or a human error in system maintenance, cannot compromise the application. Good software engineering, development practices, and system operation are important, but clearly are not enough by themselves.

It is useful to note that the security and reliability properties have some time-dependent attributes, such as the following.

Necessary time-dependent security properties:

- **Real-time availability** (including the system, data, and other resources). Ensuring that real-time processing can be done in a timely way, protecting against maliciously or accidentally caused delays.

- **Real-time accountability** such as anomaly detection and misuse detection (e.g., [177, 275]

Necessary time-dependent reliability properties:

- **Timely detection and correction of deviant system behavior,** including reconfiguration in the face of nontolerated faults or penetrations. Recovery from serious outages may or may not be allowed to incur long time delays or human intervention. Whenever human intervention is not possible, thorough advanced planning is necessary.

- **Real-time accountability** such as anomaly detection and misuse detection extended to survivability, reliability, and fault tolerance.

Necessary performance properties:

- **Functional timeliness,** such as strict bounds in hard-real-time systems, or best-effort intentions in fuzzy-real-time systems.

One of the major challenges of system development and operation is to understand *a priori* all the relevant requirements, as well as their implications on lower system layers (including hardware and communications), and to organize the system development accordingly.

Further issues on which survivability depends include human behavior — on the part of system designers and implementors, operators, users, and maintainers, for example — and acts of God. However, these may be anticipated to a considerable extent by suitable system design and operation with respect to security, reliability, and performance considerations.

In any particular application, certain vulnerabilities may not exist, or some of the threats that could expose those vulnerabilities may not exist, or the risks may be deemed inconsequential. In such cases, the survivability problem may be simplified somewhat. In general, however, it is very dangerous to base simplifications in system design, implementation, or operation on assumptions that may not be valid in practice. Therefore, great care must be taken to provide adequate assurance in any efforts that are permitted to ignore one or more of the foregoing necessary properties.

3.3 A System View of Survivability

Survivability is meaningful primarily as an emergent property of an entire computer and communication system complex, or, more broadly, of a collection of computer-based applications. Survivability also transcends lower-layer policies relating to subsystem reliability, integrity, and the like. Certain aspects of survivability are meaningful with respect to hardware as well.

Some of the properties on which application survivability typically depends are illustrated in Table 3.1, under the approximate headings of security, reliability, and performance. These three headings partially overlap, even among the different manifestations at each functional layer. For example, system integrity at any particular layer clearly contributes to security, reliability, and performance at higher layers. Similarly, prevention of malicious and accidental service denials at any layer clearly contributes to security, reliability, and performance at higher layers.

Human safety is very similar in its dependence on the same properties of the lower-layer functionality. Human safety is also largely an emergent property of the entire system complex, although particular aspects of safety can be considered at lower layers of abstraction. (For example, see [163, 164].)

Clearly, there are many more detailed properties of lower layers on which the system properties in turn depend. These are not shown explicitly, for reasons of descriptive simplicity.

Functional layer	**Security concepts**	**Reliability concepts**	**Performance concepts**
Users, operators, admins, ...	Human integrity, education, training, user identity	Human reliability, education, training, human interfaces	Human responsiveness, ease of use, education, training
Application software (SW)	Application integrity and confidentiality	Functional correctness, redundancy, robustness, recovery	Application availability, real-time performance, functional timeliness
Middleware (MW: DBMS, DCE, CORBA Webware)	SW integrity and confidentiality in DCE, Webware, DB access controls	Functional correctness, redundancy, DB backup, robustness, recovery	Functional timeliness of Web, remote DBs, and file servers
Networking (Netware)	Netware integrity, confidentiality, availability, node nontamperability, peer authentication, especially wireless	Netware integrity, error correction and fault tolerance in transmission and routing, especially in wireless roving	Netware throughput and nondenial of service, alternative routing and other infrastructural factors, especially bandwidth
Operating system (OS)	OS integrity, data confidentiality, nondenial of service, OS nontamperability, OS development and maintenance, OS user authentication	OS integrity, fault tolerance, sound asynchrony, archiving/backup OS development and maintenance	OS integrity, nondenial of service, avoidance of deadlocks, performance optimization, OS development and maintenance
Hardware (HW)	Access controls, protection domains, HW nontamperability, configuration control, protection against intentional interference, HW development	HW fault tolerance, instruction retry, error-correcting codes, HW correctness, protection against accidental interference, HW development	Processor/memory speed, communication bandwidths, contention control, adequate HW configuration, protection against any interference, HW development

Table 3.1: Some Survivability Attributes at Different Logical Layers

3.4 Mapping Mission Requirements into Specifics

Before attempting to carry out an architecture and its implementation, a vital preliminary step is to map the overall mission requirements into a specific subset of the generic requirements for survivability and its subtended attributes. This is at present a human endeavor, although the existence of computer-aided analysis tools can be contemplated to assist in requirements analysis, and subsequently to assist in determining the sufficiency of the architecture with respect to the chosen requirements.

The mapping process should take into account expected vulnerabilities, threats, and risks. It should anticipate the needs of the full range of expected applications and capabilities. The needs for each of the following functional capabilities should be anticipated from the outset, rather than discovered later in the development, and specific requirements defined.

- Multimedia facilities, including data, voice, high-bandwidth quality audio, passive or interactive video, other modes of real-time graphics, Internet telephony
- Broadcast and multicast applications
- Interoperability requirements, including open standards
- Inter- and intrasystem connectivity — for example, shared-memory multiprocessors, locally networked systems, direct Internet access, firewalls, or alternatively isolated and dramatically restricted use (as in compartmented multilevel-secure enclaves)

Chapter 4

Systemic Inadequacies

Many deficiencies in existing subsystems, systems, and networks seriously hinder the attainment of survivability. We identify those that are fundamental, and recommend specific approaches to overcome those inadequacies.

In this context, survivability is considered in the broadest possible sense, encompassing measures to handle all realistic threats — including, for example, hardware malfunctions, software flaws, accidental and malicious misuse, electromagnetic interference, acts of God, and other occurrences that are typically unanticipated. We seek to provide a realistic architectural bridge across the gap that exists between (on one hand) the present status quo of inherently incomplete requirements, criteria, standards, protocols, components, and systems, and (on the other hand) the need for survivable systems and networks that can be rapidly configured out of off-the-shelf commercial products, specifically tailored to particular applications. Unfortunately, many systems that exist today or are foreseen for the near-term future are likely to be inadequate for these purposes.

This chapter identifies some of these shortcomings, including technological deficiencies (Sections 4.1 and 4.2) and other problems (Section 4.3). Chapter 5 presents recommendations for overcoming those limitations. Chapter 8 continues the identification and analysis of the deficiencies in the context of survivable architectures, and presents specific recommendations for alternative system and network architectures, new system components, fundamental changes in how systems are developed, and guidelines for implementation.

4.1 System and Networking Deficiencies

With the almost total dependence of the U.S. Government, critical infrastructures, and the public sector on commercially available off-the-shelf systems, subsystems, and networks, all survivability-critical applications are seriously at risk.

- **Operating systems.** Existing personal-computer operating systems are seriously lacking in security, reliability, and high availability. For the most part, these systems crash frequently (sometimes very painfully, such as when an entire file system is wiped out) and offer almost no sound support for networking or multiprocessing. Minicomputer systems are only a little less worse. Supercomputers by themselves pay essentially little or no attention to security and availability concerns. Distributed operating systems are particularly vulnerable, being prone to synchronization and timing problems, deadlocks, crashes, denials of service, and serious misuse. (For example, see [67].) The absence of a so-called trusted path from a user to a trustworthy system is itself a serious failing of most systems. (Achieving suitable trusted paths is considered in Section 8.3.5.) On the whole, existing operating systems are not secure as delivered; even with very careful initialization and configuration management, many vulnerabilities are still likely to remain. The CERT Coordination Center (part of the Software Engineering Institute at Carnegie-Mellon; see `http://www.cert.org`) and other such official organizations (such as the Lawrence-Livermore CIAC, `http://www.ciac.org/`) provide a steady flow of reports on vulnerabilities and fixes (with emphasis on the latter rather than the former). Greater emphasis on flaws is provided by the BugTraq distribution (BugTraq@netspace.org), which generally gets the knowledge of vulnerabilities out long before the CERT, and which is widely read by the underground cracker community. See also NTBugTraq (http://www.ntbugtraq.com/) for Windows NT related problems.

- **Servers.** The state of the art with respect to servers is only a little better than for operating systems in general. Systems such as firewalls are typically vulnerable to misuse or to permitting dangerous events to pass through, and in actuality may themselves be risky — in part because of the illusion that they solve a larger portion of the security problem than they actually do, and in part because they themselves may be flawed. File servers and authentication servers are still not adequate for the needs of critical-survivability applications.

- **Distributed heterogeneous backup systems.** Survivable environments depend critically on the availability of facilities that perform regular trustworthy backup whenever any critical system state information changes, and rapid trustworthy recovery whenever required. In highly heterogeneous environments, the backup systems must be cognizant of the idiosyncrasies of the individual systems and servers for which backup and retrieval must be performed.

- **Systemic authentication.** Some of the most serious security deficiencies in user operating systems, servers, and networks involve the lack of adequate authentication (including authentication of users, systems, and subsystems) and the lack of trusted paths from users to systems and from systems to users. In almost all commercial systems, there are almost no guarantees that a user is interacting with the intended system. One of the most important advances in overcoming several of the deficiencies noted here involves a serious effort to improve the authentication process and the pervasiveness with which it is used throughout. Systemic authentication and integrity of resource management are also considered in Section 8.3.4.

- **Networking protocols.** The various Internet protocols at different layers in the protocol stack (e.g., FTP, Telnet, and SMTP at the application layer, UDP at the transport layer) and their implementations (e.g., `sendmail` as an implementation of SMTP) are for the most part seriously deficient when it comes to security, and weak with respect to reliability. Unfortunately, they are very widely used, despite their glaring weaknesses. Various successors to the IP Internet protocol suite have been proposed, although their realistic strengths and weaknesses remain to be seen. IPSEC (`http://www.ietf.org`) is intended to provide end-to-end encryption over the Internet IP Version 4, but only minimal authentication and integrity. Various weaknesses of IPSEC have been analyzed (e.g., [34]). IP Version 6 is intended to provide a more robust set of protocols. Domain Name System Security (DNSSEC) protocols have also been proposed, with a public-key infrastructure independent of the DNS-domain public-key infrastructure. Among its other known vulnerabilities, DNSSEC is vulnerable to compromises of its all-powerful hierarchical root. Although further study and refinement are needed, these proposals seem to be aimed at only a small portion of the security problem, and a much smaller portion of the overall survivability problem. For those environments in which fixed communication facilities are not possible or desirable, the requirements and facilities for secure, reliable, and survivable wireless communications are still in their infancy.

For background on TCP/IP and related protocols, see the massive collection of Requests for Comments (RFCs). The RFCs are a goldmine of information on Internet protocols. For example, the latest draft security architecture for the Internet Protocol is RFC 2401, dated November 1998 (`ftp://ftp.isi.edu/in-notes/rfc2401.txt`). This provides security services for the IPO layer in IP versions 4 and 6.

GloMo (`http://www.darpa.mil/ito/research/glomo/index.html`) — the DARPA Global Mobile effort — is seeking to address some of the basic engineering issues in research and prototype developments, and is also attempting to use Fortezza technology for cryptographic approaches to retrofitting security into the GloMo research environment (`http://www.glomo.sri.com/`). However, achieving secure portable computer communications is a very difficult task, and at present security and to a large extent survivability are not driving requirements in most of the ongoing DARPA GloMo research program; rather, security (not to mention survivability) seems to be thought of as something that can be added later — which goes against the teachings of years of experience in system development.

- **Cryptographic protocols and embeddings.** Although strong cryptographic algorithms exist, their implementations tend to be vulnerable to compromise. Because of U.S. Government policies regarding strong cryptography and because of the inherent risks associated with key-recovery schemes, the use of cryptography is far less prevalent than is needed to support meaningful security. (For example, see [6, 7, 79, 227].) There are surprisingly many ways of breaking cryptographic embeddings without exhaustive key searches and without breaking the algorithm — such as Paul Kocher's timing attacks [148], Kocher's differential power analyses (Risks Forum, volume 19, issue 80, `http://catless.ncl.ac.uk/Risks/19.80.html`), or the fault-injection techniques of Anderson and Kuhn [14] and others [24]. However, even brute-force attacks are becoming feasible, as demonstrated by the Electronic Frontier Foundation sponsored machine, Deep Crack [99] — which is capable of finding DES keys, on the average after just a few days of exhaustive search (at the cost of less than a quarter of a million dollars to build the initial Deep Crack machine). More recently, the 1999 RSA DES Challenge was broken in less than 24 hours by Distributed Crack, a consortium of many different computers (including Deep Crack, which happened to be the machine that actually discovered the Challenge key, after a random search of only 9% of the key space; Distributed Crack itself had exhausted 23% of the key space at that time, indicating that the aggregate power of the systems other than Deep Crack was considerably greater than Deep Crack itself). The marginal cost of Distributed Crack is therefore essentially zero, because it is merely using free cycles from participating systems anywhere on the Internet.

Schneier and Mudge [315][1] have uncovered an astounding variety of elemental flaws in Microsoft's implementation of the Point-to-Point Tunneling Protocol (PPTP), only a few of which have apparently been fixed. Among the RFCs, several recent ones are worth noting:

- RFC 2408 (`ftp://ftp.isi.edu/in-notes/rfc2408.txt`) is the Internet Security Association and Key Management Protocol (ISAKMP).
- RFC 2409 (`ftp://ftp.isi.edu/in-notes/rfc2409.txt`), the Internet Key Exchange (IKE), provides authenticated keying material for use with ISAKMP.
- RFC 2412 (`ftp://ftp.isi.edu/in-notes/rfc2412.txt`), the OAKLEY Key Determination Protocol, is based on the Diffie-Hellman key exchange algorithm.
- RFC 2411 (`ftp://ftp.isi.edu/in-notes/rfc2411.txt`), *The IP Security Document Roadmap,* is helpful in relating the various documents to one another.

Of particular importance is the need for robust public-key infrastructures (PKIs) that compatibly provide public-key certificates and validation of the genuineness of the certificate authorities, with sufficiently good performance, to make cryptographically protected interoperability practical. Many commercial distributed PKIs are emerging, and are expected to win out over a few highly unified PKIs. However, compatibility and interoperability are still badly lagging.

- **Computer-communication infrastructure.** The Internet, routers, IP addresses, domain name servers, network information services, and other infrastructural components are themselves vulnerable to misuse and accidental failures.
- **Hardware.** With the prevalance of RISC (reduced instruction set computers) hardware architectures, some of the security mechanisms have disappeared from types of processor designs that once paid serious attention to security. Even in cases in which domain separation is provided in the hardware, operating systems tend not to use it effectively. Similarly, some of the more advanced and powerful techniques for reliability and fault tolerance are typically not used in commercial hardware. In the stand-alone personal-computer world, occasional hardware crashes are not considered important events. Unfortunately, such hardware is often found in critical systems.

---

[1] Mudge is part of a White-Hat Hacker group known as L0pht. L0pht's 1998 Senate oral testimony [205] claimed that it is possible to bring down the entire Internet in about 30 minutes.

- **Real-time monitoring and analysis.** At present, real-time analytic environments tend to be special purpose — for example, focusing on attempted exploitations of a few known security vulnerabilities or a few known threats to reliability and availability. It is difficult today to provide real-time monitoring and analysis with respect to suspicious but unspecific events, and with respect to the simultaneous satisfaction of multiple requirements. It is also difficult to handle highly distributed environments, where correlation of different analytic results would be advantageous. Furthermore, the absence of meaningful authentication and accountability makes it very difficult to determine the source of security intrusions.

4.2 Deficiencies in the Information Infrastructure

The previous section outlines numerous deficiencies in computer systems and in networking software. However, some fundamental problems transcend deficiencies in systems and networking software - specifically, those relating to the underlying information infrastructure. We are becoming critically dependent on the Internet, and are likely to be even more dependent on whatever succeeds it.

Unfortunately, the Internet has become an enormous self-perpetuating organism of its own — with no coherent management, no overall control, and almost no ownership by any national or corporate entities (except in nondemocratic countries). Its existence is almost totally unregulated, and it is run on the fly in a strikingly unprofessional way. Because traffic may be routed arbitrarily through potentially untrustworthy and unreliable nodes, retrying over alternate routes is at present typically the best that can be done when problems are experienced. But that approach is vulnerable to massive denial-of-service attacks. If a vital gateway is down, an entire enterprise may be off the Internet. In general, weaknesses in the infrastructure become weaknesses in the computer systems and networking software. For example, the fundamental deficiencies in networking protocols noted in Section 4.1 affect not just local networks and enterprise-internal networks, but also the Internet; they are likely to haunt any future information infrastructures unless radically superseded. Conversely, weaknesses in computer systems and networking software can result in weaknesses in the information infrastructure — affecting telecommunications and power distribution as well.

4.3 Other Deficiencies

- **System development practice.** The system-development practice in the U.S. is not good, and in many cases could be called disastrous. Good system engineering and software engineering practice is extremely rare. Many U.S. Government systems have had to be abandoned (e.g., the FAA Air-Route Traffic Control System, the FBI fingerprint system, and the IRS Tax System Modernization effort) after the expenditure of billions of dollars. The record of states and private corporations is not much better (e.g., the California Statewide Automated Child Support System known as the Deadbeat [Moms' and] Dads' Database). Systems that are actually completed and deployed are often riddled with security and reliability flaws and performance problems. Numerous examples are given in [228]. (See also James Paul's Congressional report [262] on some of the difficulties of software development.) The Year-2000 Problem is of course a massive and pervasive example of very bad system development practices that have persisted for many years.

System developments fail for a wide variety of reasons, including inadequately specified requirements, inadequately specified system designs, poor implementations, poor documentation, poor management, and poor choices of programmers. Projects that have many programmers seem to be much less likely to succeed, although extremely gifted management and skilled team-oriented programmers can make such efforts successful. (However, there are very few success stories.) Projects with inadequate staffing or misassigned personnel can also be seriously impaired. Overcommitted contractors and improperly guided subcontractors are also sources of development risks.

- **Problems with proprietary products.** One of the most serious obstacles to survivable systems and network environments involves the use of proprietary standards and proprietary products, which can make interoperability with other systems very difficult. These difficulties arising from proprietary systems and interfaces can be further complicated by developers' desires to make their own new systems backward compatible with their older systems. These obstacles often result in earlier deficiencies being perpetuated into future systems. Furthermore, short-sighted commercial developers seem to have little interest in making their products interoperable with less widely used products from other vendors. Perhaps most important, proprietary systems cannot be subjected to the same scrutiny that open-source systems can. This has always been of particular importance for security, for which peer review, design analysis, code analysis, and in some cases formal methods can play an enormous role in detecting system flaws. The same argument applies to survivability and its other subtended requirements as well.

- **System procurement practice.** Would-be acquirers of systems with critical survivability and security requirements often have an even more difficult task than the system developers. In many cases, the requirements to be met are not adequately understood, or are inadequately specified. Even when the requirements are well established, no off-the-shelf solutions are likely to exist, which necessitates additional software — which can result in further development problems. Furthermore, procurements tend to favor popular products, irrespective of whether they are robust.

- **Understanding of the vulnerabilities, threats, and risks.** There is an enormous lack of understanding on the part of system procurers, government officials, developers, system procurers, users, system administrators, and users. Unfortunately, critical requirements for survivability and security tend to necessitate greater understanding on the part of these individuals, and also put extreme stresses on computer systems, networks, and people alike.

- **High-performance computing and communication** (HPCC). The field of supercomputing has been slowly transforming itself in recent years. For decades, it had remained a highly overspecialized cult, with inordinate emphasis on special-purpose processors, the use of assembly code in the hopes of program optimization, and the use of compilers for Fortran and other languages that were inherently ill suited for the intended applications. The field largely ignored reliability and fault tolerance in hardware. It demonstrated very little interest in mainstream concepts — such as multiprocessor architectures or networked configurations of systems, and most of computer science — including robust algorithms for concurrency, synchronization, deadlock avoidance. Indeed, it had an almost total disdain of what software might be able to contribute — ignoring operating systems, compiler generators, *make* files, preprocessors, translators, and different approaches to programming languages other than developing parallelizing compilers for languages not designed for such use (e.g., Fortran). Application programming has suffered from the inability to flexibly use widely available resources, and has also been plagued with overreliance on flawed or suspect simulation models.[2] As systems with stringent survivability requirements increasingly seek to employ supercomputers, such problems are likely to be exacerbated. Fortunately, a few efforts to modernize the field are finally beginning to emerge — including the DoD High Performance Computing Modernization Program (`http://www.hpcmo.hpc.mil`). Of considerable potential interest is the joint UCLA-Hewlett-Packard massively parallel Teramac project, which achieves supercomputer capabilities despite the presence of hundreds of thousands of hardware defects [115]. In a rather novel approach, processor functionality is determined after implementation rather than specified beforehand, thus accommodating what would otherwise be considered to be permanent fault modes; it does what it does (although it is not clear how it handles transient faults). At a completely different point in the architectural spectrum, massive computational resources can be acquired over the Internet, for example, using Sun Microsystems' Jini, which uses Java to enable arbitrary modes of distributed multiprocessing across the Internet. Considering the power of the mobile-code paradigm (Section 8.4), that approach is likely to win out in many cases.

Although entire books can be and have been written about these deficiencies and the resulting risks (e.g., [60, 228]), the emphasis here is on overcoming the deficiencies — as addressed in the next chapter.

---

[2] See pages 219–221 of [228] for cases in which simulation models were themselves seriously faulty. See also a terse but incisive summary of problems with large-scale simulations by D.E. Stevenson [328].

Chapter 5

Approaches for Overcoming Deficiencies

> *We can't solve problems by using the same kind of thinking we used when we created them.*
>
> Albert Einstein

Given the deficiencies identified in Chapter 4, we next consider what might be done to overcome them. Various approaches toward prevention, detection, reaction, and iteration are summarized in Table 5.1 for each of the three primary attributes of survivability noted in Figure 3.1 and discussed further in this and subsequent chapters. Specific architectural recommendations are considered in Chapter 8 for systems and networks with stringent survivability requirements. Ultimately, the use of robust architectures is absolutely fundamental to the recommendations of this report.

5.1 System and Networking Architectures

Two fundamentally different architectural approaches seem possible — either increase the security, reliability, and survivability of the most critical components, or else develop system and network architectures that are survivable despite the presence of inherently weak components. In practice, a combination of both approaches is desirable, for several reasons:

- End-user personal computers, network computers, hand-held wireless communicators, and dumb terminals are likely to be lowest-common-denominator components.
- Intelligent workstations are not likely to enforce multilevel security.
- Systems and networks tend to be heterogeneous.

Indeed, component survivability can benefit greatly from component diversity. Thus, it becomes essential to identify the most critical components, and to concentrate sufficient architectural strengths in those components. The basic challenge is to considerably reduce the extent to which all subsystems must be extensively trustworthy.

Approach	Relability	Security	Performance
Prevention	Robust architecture: Redundancy: error correction, fault tolerance	Robust architecture: Domain isolation, access controls, authorization	Robust architecture: Spare capacity
Detection	Redundancy: error detection	Integrity checks, anomaly/misuse detection	Performance monitoring
Reaction	Forward/backward, recovery	Security preserving reconfiguration	Reconfiguration
Iteration	Fault removal	Exploratory off-line patches	Redesign, tradeoffs

Table 5.1: Defensive Measures

System architectures must address the necessary survivability-relevant requirements, including reliability, fault tolerance, and security — irrespective of which components are actually trustworthy with respect to which requirements. In addition, these architectures must be flexible enough to support real-time applications and supercomputing requirements, rather than necessitating special-purpose designs. Various architectural alternatives are considered in Chapter 8. The notion of multilevel survivability (Section 1.2.8) is being explored as one approach to system structure. However, it should be taken not as an attempted universal property of kernels and trusted computing bases (as was attempted with MLS), but rather as a potentially useful architectural driving force in the context of the notion of generalized dependence.

5.2 System and Networking Components

The components that are most critical for survivability — and therefore deserving of the most defensive design, development, and maintenance attention — are typically authentication servers, file servers, network servers, boundary controllers (including access control mechanisms, firewalls, guards), as well as other components that must necessarily be at least partially trusted. From a reliability point of view, file servers and network servers are particularly critical. From a security point of view, authentication servers, boundary controllers, and cryptographic units must receive extra protection. From a system integrity point of view, cryptographically generated integrity seals and proof-carrying code are useful techniques to hinder undetected system modifications.

Particularly vital are the trustworthiness of cryptographic protocols and the dependability of their implementations. See a recent report by Abadi and Gordon [2] as part of a series of papers on formalizing cryptographic protocols (they include copious references to earlier work), along with further discussion in Section 5.8. Composability of such components is considered in Section 5.7.

5.3 Configuration Management

Several considerations are necessary to be able to readily configure survivable systems and networks out of subsystems. Architecturally, the subsystems must have appropriate functionality. They must also be compatible with one another, and their interfaces must be easily composable. System and network management facilities are also critical to the maintenance of survivability. In particular, the configuration management system must be both comprehensive and comprehensible enough to permit consistent administrative control.

5.4 Information Infrastructure

The previous sections of this chapter outline improvements that need to be made in computer systems and in networking software. However, Section 4.2 notes that some fundamental problems cannot be solved through better systems and networking software — specifically, those relating to the underlying information infrastructure (such as the Internet as it exists today). Overcoming the limitations of the Internet is an enormous undertaking, but some drastic measures must be taken immediately to prevent those limitations from becoming significantly worse. The biggest problem of course is that the Internet is an international entity. In May 1996, at a hearing of the U.S. Senate Permanent Subcommittee on Investigations (Senate Committee on Governmental Affairs), in light of testimony on difficulties that exist with being connected to the Internet, Senator Sam Nunn ([340], pages 10-11) asked in essence what would happen if we (the United States) simply cut ourselves off from the rest of the Internet. Perhaps having a national computer-communication infrastructure in addition to the international Internet is in fact a good idea, although it would not solve the problem that the computer systems and networking software are not secure enough, and would defeat the global information interchange purposes of the Internet. But even more fundamentally, if it had gateways and dial-up connections that could be accessible from the rest of the world, it would be very difficult to seal it off. Nevertheless, rigidly controlled private networks are clearly a good idea. (See also [230, 232, 205] for subsequent relevant Senate testimonies.)

This leads us to one of our most far-reaching conclusions on overcoming the existing deficiencies. It is urgently necessary to supplant the existing TCP/IP/ftp/telnet/udp/smtp set of protocols. Ideally, a fundamentally new set of protocols could be engineered to provide the necessary survivability, security, reliability, and performance (all at once), with robust authentication as a fundamental requirement. Alternatively, a few of the existing protocols might be successfully modified, but we are not encouraged at this point at the likelihood of small incremental improvements. Although IPSEC and IP Version 6 attempt to overcome some of the most glaring weaknesses, they are still not strong enough. In certain less critical cases, it might be possible to use a subset approach, parameterizing the protocols accordingly. However, in the long run, the strategy of replacing the existing fundamentally defective protocols with a new set of survivable and secure protocols might actually be less costly than trying to coexist with those protocols that are clearly not up to the job. In that way, it would be possible to develop highly survivable separate information structures, with perhaps some possibility of trustworthy but highly controlled interoperability with the rest of the world (e.g., the Internet).

5.5 System Development Practice

In general, system development practice is truly abysmal and must be improved dramatically. If systems are to be configured primarily out of off-the-shelf components, then development practice is vital to the dependability of those components. However, it is not realistic to expect that operating systems and networking software will improve dramatically. On the other hand, once subsystems have been developed, it is too late to quibble about bad development practice — particularly if completed systems and networks are to be assembled rather than developed. Furthermore, the best development practice is not very effective if the basic architecture is not suitable.

A recent thoughtful but somewhat simplistic article by Paul Green [110] is worth noting, entitled "The art of creating reliable software-based systems using off-the-shelf software components." Although the article is concerned primarily with application-system reliability, it presents a few practical guidelines based on Green's 17-year experience at Stratus Computer. For example, if you are a procuring agent, he suggests (we oversimplify for descriptive purposes) that you should select vendors who are committed to reliability, insist on good software engineering practice throughout, make sure your contracts cover the entire life cycle, test in the large and force that process on your contractors, and don't pay off vendors until they deliver what they are required to produce (assuming that you have specified the requirements adequately in the first place).

5.6 Software Engineering Practice

Good software engineering practice involves the use of modular design, functional abstraction, well-specified functionality, reusable and interoperable interfaces, information hiding to mask implementation detail, and the use of analysis techniques and tools that greatly reduce the likelihood of flaws and programming errors. Good software engineering practice therefore should also involve the use of high-level programming languages that are intrinsically less susceptible to characteristic errors — such as missing bounds checks, mismatched types and mismatched pointers, off-by-one errors, and missing exception conditions. As widely used as the C programming language is, it is a continual source of programming errors by skilled programmers as well as novices. Careful documentation is also essential. Potentially, the most powerful techniques may in the long run involve formal methods (Section 5.8), but those techniques are labor intensive and are now becoming much more effective in practice.

Commercial techniques for software engineering tend to emphasize procedures for controlling and constraining the *processes* involved in the software development cycle: requirements engineering, specification, implementation, testing, management, quality assurance, and risk management. Testing is inherently incomplete, with the old adage that it demonstrates only the presence of bugs rather than the absence of bugs. Metrics are popular for the development process and for assessing code quality, but are not definitive. Code inspections are also popular, but not conclusive. Assessment of risk management is inherently a risky process (see [228], pages 255–257).

Many of the commercial software engineering techniques are supported by automated or semiautomated tools. Indeed, the ones that are not supported by mechanical tools are of very limited value. The use of software engineering tools can be advantageous, particularly in detecting the characteristic errors noted above. One of the most useful tools in recent years is the `purify` program, which detects garbage collection problems resulting from unfreed storage. However, overreliance on such tools can result in serious risks in the absence of human intelligence. Furthermore, overemphasis on the processes rather than on the requirements, the designs, and the implementations themselves can be misleading. Not surprisingly, the Year-2000 Problem is forcing a rethinking of much of the old-style conventional wisdom relating to software engineering. Those who take the challenge seriously are likely to realize the need for radical change in making the so-called software engineering field more of an engineering discipline. (See a provocative article by David Parnas [256] on that subject.)

Use of the object-oriented paradigm in the system design itself may be beneficial (particularly with respect to system integrity), for example, creating a layered system in which each layer can be looked upon as a strongly typed object manager, as in the PSOS design [94, 235]. That paradigm combines four principles of good software engineering – abstraction, encapsulation, inheritance, and polymorphism. Inheritance is the notion that related classes of objects behave similarly, and that subclasses should inherit the proper type behavior of their ancestors; it allows reusability of code across related strongly typed classes. Polymorphism is the notion that a computational resource can accept arguments of different types at different times, and still remain type safe; it permits programming generality and software reuse. (A recent effort to model dynamically typed access controls is given by Tidswell and Potter [336].)

Ultimately, the choices of which of many development methodologies, testing techniques, and assurance methods are used is not particularly critical. What is most important is that software development managers understand the development process, that designers understand the full implications of their designs, and that implementors respect the integrity of the designs when those designs are adequate but that they also recognize when the designs are faulty. The methods and tools can go only so far. Inevitably, it is the people in the process that matter, and they cannot be automated.

Software Architecture in Practice [27], a potentially useful recent book from the Carnegie-Mellon University Software Engineering Institute, considers the business cycle and organizational forces behind software architecture. It presents a management-oriented view of some of the problems that we consider here.

5.7 Subsystem Composability

Parnas [72, 251, 250, 252, 260, 261, 253, 258, 254, 257, 255, 259] (listed chronologically) and Dijkstra [87, 88, 89] have for many years written extensively on the modular decomposition of system designs. (Various other authors have written more recently on this subject.) Unfortunately, most commercial systems are seriously lacking in their architectural structure.

The effects of module composition on the corresponding security models have been studied extensively in recent years (e.g., [3, 36, 120, 180, 183, 184, 185, 186, 189, 196, 295, 299, 333, 354, 355]). In many cases, however, seemingly straightforward compositions have unpredicted side effects, in some instances interfering with one another. (A case of supposedly independent cryptographic security protocols interacting unsecurely is given by [143].)

In other efforts less specifically related to security, there has been considerable research in combining models, theories, equational and other logics, term-rewriting systems, and data structures. However, most of these efforts have considered the simplest forms of composition (particularly those involving serial hookups without feedback) and the effects that result from simple combinations of policies or models; these efforts have often been extremely theoretical — with not much applicability to real system needs such as the implications of composed implementations.

When great care has been taken to achieve interoperable modularity, modular composition is relatively straightforward. More commonly, however, composition is not an *a priori* design consideration, and composability may be very difficult. Ideally, it should be possible to configure a specific system capable of attaining the desired (sub)set of requirements, parametrically tailored to each specific application. Integration of the chosen components should be attainable with minimum effort, with respect to design, implementation, operation, and maintenance. Composition should address the incorporation of less trustworthy components (e.g., as in Byzantine agreement) as well as compromise of trustworthy components. As applicable, common useful middleware components should be identified from which survivable systems can more readily be configured — in the sense of a virtual survivable trusted computing base that can survive certain threats despite the presence of a less trustworthy underlying operating system. System adaptability under perceived threats may also be useful.

At the Eighth ACM SIGOPS European Workshop in Sintra, Portugal, in September 1998 (see [23]), a rather awkward debate took place. The stated argument was that the development of robust distributed systems from components is *impossible* [145]. Although "a marginal majority disagreed" with this proposition, there are strong arguments that emerge from the discussion bearing on why composition is not straightforward in any realistic situations. However, a deeper conclusion that might be drawn from that debate is that we must work much harder to establish criteria under which composition does not compromise robustness, and perhaps even enhances it — as suggested by the notion of generalized dependence.

5.8 Formal Methods

> *[T]he representation of structure is the most important aspect of programming for purposes of formalization.*
> Bob Barton, May 1963 [26]

Formal methods can play an important role in the attainment of systems and networks that must achieve generalized survivability, in specification, design, and execution. Great improvements in system behavior can be realized when the requirements (such as survivability, security, and reliability) have a formal basis. Similarly, enormous benefits arise whenever design specifications have a formal basis — especially if they are derived from well-specified requirements rather than the common practice of being established after the fact to represent an ad hoc assembly of already-developed software (sometimes referred to as putting the cart before the horse). Formal design verification then involves formal demonstrations that the specifications are consistent with their requirements, providing no less than required — and to the extent that the absence of Trojan horses can be demonstrated, nothing unexpected that might be harmful. Verification of designs is difficult for systems that were not designed to be readily analyzed, but can nevertheless be valuable in legacy systems (as in analyses of the risks associated with the Year-2000 problem). Finally, although it is less commonly practiced in software, formal code verification can demonstrate that a given implementation is consistent with its specifications. Formal hardware verification is being used increasingly, and demonstrates the potential effectiveness of formal methods where there are considerable risks (financial or otherwise) of improper design and implementation.

Various formal methods can be valuable in specifying and analyzing requirements, designs, and implementations, as well as in compositionality. Of particular importance in connection with survivability are techniques that can provide formal relationships between different layers of abstraction — with respect to requirements and specifications alike. The use of formal methods is recommended in particularly critical applications, and can help move the current highly unpredictable ad hoc development process into a much more predictable formal development process. In the long run, use of such techniques can dramatically decrease the risks of system failure. Contrary to popular myth, judicious use of formal methods can also decrease the overall development and operating costs — especially when the costs of aborted developments (such as the cancellations of the IRS, FAA, FBI systems noted in Section 4.3) are considered, along with the costs of overruns, delays in delivery, and subsequent maintenance.

Judicious use of formal methods can have a very high payoff, particularly in requirements, specifications, algorithms, and programs concerned with especially critical functionality — such as concurrency, synchronization, avoidance of deadlocks and race conditions in the small, and perhaps even network stability and survivability in a larger context, derived on the basis of more detailed analyses of components. There is no substitute for using demonstrably sound algorithms (e.g., [311]).

Important early work on the effects of composition using hierarchically layered abstractions was part of SRI's Hierarchical Development Methodology effort (see Robinson and Levitt [292]). For some reason, this work is still relatively obscure, although it is vital to formal reasoning about subsystem composition and analysis of emergent properties.

Of particular importance is the formal analysis of requirements — for example, determining whether a given set of requirements at a particular layer of abstraction is consistent within itself, whether the different sets of requirements at the lower layer are fundamentally incompatible with one another, and whether the requirements at a lower layer are consistent with the requirements at the upper layer. Once such an analysis is done, then it is also beneficial to determine whether system specifications and implementations are consistent with the relevant requirements. (An application to safety requirements is given in [117].)

It must be emphasized that the most valuable uses of formal methods are in finding flaws and inconsistencies, not in attempting to prove that something is completely correct. However, formal methods approaches are not absolute guarantees, because problems can exist outside of their scope of analysis. For example, suppose that a given analysis does not detect any flaws or inconsistencies in a specification or implementation. It is still possible that the requirements are inadequate (e.g., the specifications could fail to prevent a problem not covered by the requirements), or that the analysis methods themselves could be flawed. For these reasons, extensive testing of developed systems is also important – albeit inherently limited.

Unfortunately, testing is itself inherently incomplete and incapable of discovering many types of problems — for example, stemming from distributed system interactions and concurrency failures, subtle timing problems, unanticipated hardware failures, and environmental effects. Exhaustive testing over all possible scenarios is basically impossible in any complex system.

Considering that survivability, security, reliability, and fault tolerance are all weak-link properties, formal methods and nonformal testing are both useful approaches in attempting to find the weak links. Neither is adequate by itself. An interesting nonformal approach to fault injection to detect failure modes is given by Voas and McGraw [344]; similar ad hoc approaches are common with respect to red-team attacks in testing would-be secure systems.

Formal methods have been used extensively in the past for security (e.g., [57, 74, 134, 162, 165, 229, 242, 243, 279, 300], fault tolerance (e.g., [63, 122, 153, 169, 192, 206, 207, 247, 265]), general consistency [93], object-oriented programs (e.g., [4]), composability (as noted in Section 5.7), compiler correctness (e.g., [331]), protocol development (e.g., [123, 144, 323]), hardware verification and computer-aided design [326], and human safety (e.g., [117, 309]) but to our knowledge not for survivability, or for security and fault tolerance in combination. One serious attempt at a broader approach comes from the European dependability community, which tends to consider dependability as an all-embracing quality (as noted in Section 1.2.3). A representative example of that approach is found in the work of Gerard Le Lann [160, 161] relating to "X-critical applications", where X could be any qualifier such as *life, mission, environment, business,* or *asset,* although his formal methods have thus far been applied primarily to fault tolerance. See the discussion of the role of formal methods in secure system architectures by Neumann [229].

Under the present project, Jon Millen is studying earlier security-related work of Catherine Meadows, and is generalizing it to address the configurability of survivable services. Some of the results of his work to date are given in two research papers [197, 198] that characterize reconfiguration as a kind of flow property that can be formally satisfied. (Jon Millen is also working under a DARPA contract to formally model and prove properties of network protocols and cryptographic protocols, using SRI's PVS verification system for formal proofs – `http://www.csl.sri.com/pvs.html`.)

Millen's survivability measure paper [198] in progress extends the system model introduced in the reconfiguration paper [197] to a structural hierarchy. Components of system services are viewed as services with components of their own. With this additional dimension, one can define dependency of a service on a lower-level service, and look for a lattice-valued measure of survivability for comparing services that may be at different levels. The concepts in the measure paper have been simplified dramatically, yet still lead to a max-min formula for the measure that satisfies the intuitively necessary properties. Further development is planned in the second phase of the project to establish uniqueness properties for the measure and to study other properties of the hierarchical structure, such as criticality of sets of components.

Formal methods are also the basis for methods for belief logics that permit the systematic analysis of cryptographic protocols, stemming from the Burrows-Abadi-Needham BAN logic [62], as well as more recent work by Gong, Needham and Yahalom [106], Meadows [190], Kailar and Gligor [136], Alves-Foss [9], Abadi and Gordon [2], and others. There is considerable other work on formal analysis of cryptographic protocols, including Meadows [190] on key management, Lowe [172] on Needham-Schroeder, and Paulson [263, 264], Mitchell et al. [200], and Lincoln et al. [166] on cryptographic and authentication protocols in general. See also Abadi and Needham's formulation of prudent engineering practice for cryptographic protocols [5].

Bellovin [33] shows how formal verification can be used to constrain the code generation process, which can be particularly important in open-source compilers, where consistency between the semantics of the source code and the semantics of the object code is critical, independent of the compiler.

Formal methods can also be used in execution, as in proof-carrying code that can be used to ensure that a critical component has not been tampered with. For example, see George Necula's thesis work [213] and Web site (`http://www.cs.cmu.edu/~necula/`), including a hands-on demonstration. (An earlier one-page summary of Necula's work in progress is given in [214].)

Of particular interest in the context of highly survivable systems, formal methods have a potentially vital role in robust open-source software, considered in Section 5.9.

The SRI Computer Science Laboratory formal methods Web site has assembled an extensive collection of URLs (see `http://www.csl.sri.com/pvs.html`) representing work within CSL and elsewhere in the world on formal methods.

5.9 Open-Source and "Free" Software

In contrast to the proprietary software that is the lifeblood of most commercial interests, a variety of nonproprietary forms of software exists, including what is sometimes called open-source software, free software, or simply nonproprietary software. These designations actually have subtle differences, and cause considerable confusion over their meanings. However, all of them attempt to give the users certain freedoms that are impossible with proprietary software. In particular, *open-source* software implies that the source code is available. (Restrictions on the use of that term are imposed by the trademarked Open Source Definition — but not restrictions on the software — as noted below.) In contrast, *free software* implies that the users and redevelopers of the software have certain freedoms that do not arise with proprietary software; however, the cost is not necessarily free and there are still opportunities for entrepeneurs in developing and maintaining such software.

Because some of the serious systemic deficiencies are not likely to be overcome in proprietary systems (Section 4.3), it would be highly advantageous to make more systematic use of nonproprietary software, especially if the source code is openly available, and if it can be made more robust than its proprietary counterparts, and if trustworthy distribution paths can be established and used consistently in a trustworthy manner. Also important is the systematic use of nonproprietary interface standards that have been explicitly created with interoperability in mind.

Dependence on closed-source proprietary code can have many disadvantages:

- Unavailability of source code leads to inscrutability, which means that the code is unanalyzable in any practical sense, particularly by knowledgeable colleagues who might find serious flaws.
- Nonmodular closed-source systems can be enormous. The inability to be subsetted typically hinders the removal of unnecessary components, and further adds to inscrutability.
- Closed-source systems often suffer from a serious lack of interoperability and composability, which in turn can result in systems with poor survivability (as well as poor satisfaction of the subtended requirements — especially security and reliability).
- Closed-source environments tend to induce single-vendor solutions, which in turn tend to preclude constructive evolution.
- Closed-source systems cannot be adapted readily to different situations or easily and quickly repaired when serious flaws are discovered.

Windows NT 5.0 reportedly will have 48 million lines of source code (most of that appears to be kernel code), with another 7.5 million lines of associated test code. It is illustrative of each of these factors. Unfortunately, the totality of code on which survivability and security depend is essentially the kernel and operating system plus potentially all of the application code that can be loaded at any time. That represents an enormous amount of code that must be *trusted* (because it is not *trustworthy*) in any critical application. (Recall the divide-by-zero in an NT application that brought the Yorktown Aegis missile cruiser to a halt, in Section 1.6.)

In addition, dependence on proprietary interface standards greatly complicates integration of large systems out of mix-and-match components that could otherwise be well suited to the tasks at hand.

A humorous but subliminally serious assessment of the use of commercial off-the-shelf (COTS) systems is given by David Carney [65].

In contrast with proprietary closed-source software systems, open-source software, free software, and nonproprietary offer opportunities to surmount these risks of proprietary software, in various ways.

"Open Source" is registered as a certification mark, subject to the conditions of The Open Source Definition (`http://www.opensource.org/osd.html`), which has various explicit requirements: unrestricted redistribution; distributability of source code; permission for derived works; constraints on integrity; nondiscriminatory practices regarding individuals, groups, and fields of endeavor; transitive licensing of rights; context-free licensing; and noncontamination of associated software. For background, see the opensource.org Website, which cites GNU GPL, BSD Unix, the X Consortium, MPL, and QPL as conformant examples.

Extensive information relating to open-source and free software can be found on the Free Software Foundation Website (`http://www.gnu.org`) — including a treatise by GNU's Richard Stallman (`http://www.gnu.org/philosophy/free-software-for-freedom.html`) on "Why 'Free Software' is better than 'Open Source' ".

Further examples of free software are Perl (Practical Extraction Resource Language), Bind (allowing symbolic naming of IP addresses), and the GNU System, which includes Linux (which encompasses many interoperable components such as GNU Emacs and GCC). ("GNU" is a recursive acronym, representing "GNU is Not Unix".) The Netscape Web browser and the Apache Web server are hybrid examples, satisfying most of the open-source requirements, but with certain restrictions on the disposition of the source code.

It is a sad commentary on many commercial and proprietary software developments that some of the most useful, flexible, and robust software components today are open-source software products, often the results of labors of love, and widely available free of charge over the Internet or with minimal encumbrances. (Two examples of open-source software were particularly valuable in the preparation of this report: the GNU Emacs editor and the LaTeX document system.)

Of course, potential risks can be associated with nonproprietary software — for example, relating to the authenticity of the sources and the trustworthiness of the distribution paths. To combat ordinary code hacking as well as the three forms of compromise noted in Section 1.3, a broad-spectrum combination of techniques is desirable, including (for example) cryptographic checksums, trustworthy software distribution channels, and public-key authentication schemes, which together can overcome some of the uncertainty as to the trustworthiness of any code version that you might be using.

Particularly serious potential problems with Trojan horses might be implanted in variant versions of open-source software. A paradigmatic risk is provided by Ken Thompson's C compiler example [335], noted in Section 1.3. In fact, compilers used to produce critical-system code present some special problems. Bellovin's approach to using formal verification [33] is relevant in demonstrating consistency between source code and object code, which is a particularly thorny problem when insiders (such as Ken Thompson!) are able to tinker with the compiler itself.

It is unfortunate that so few robust open-source security systems exist, particularly because closed-source systems represent a violation of the principle of scrutability (see Section 8.1. In a recent communication, Stallman notes that the GNU Project is working on Free Software for public-key encryption. The GNU Privacy Guard, a free and non-patent-infringing replacement for the non-free program PGP, is already being used. LSH, a free and non-patent-infringing replacement for the non-free program SSH, is in development but not yet ready for use.

The research literature is full of public-key-based authentication protocols, and an important recent demonstration showed that serious authentication cannot be done without some form of public-key crypto [113]. The Diffie-Hellman public-key cryptographic algorithm [86] is now in the public domain. A few simple schemes for login authentication are freely available, such as S-Key one-time passwords. The MIT Athena Kerberos and Berkeley BSD Unix are further examples where security has been a serious concern. Similarly, PGP

(Pretty Good Privacy) is becoming more widespread as it becomes seamlessly embedded in e-mail environments, although has had some proprietary underpinnings. Some of those products can also be obtained commercially through organizations that provide operational and maintenance support, such as PGP and Red Hat Linux. Indeed, it is not essential that nonproprietary software be available free of charge, and considerable value can be added by commercial enterprises. What is important is that the software be available for open scrutiny, able to be improved over time as a result of an open collaborative process, and able to be subjected to distributional controls to ensure its integrity.

We need significant improvements on today's software, both proprietary and otherwise, to overcome myriad risks (see the RISKS archives, `http://catless.ncl.ac.uk/Risks/`, or the Illustrative Risks document, `http://www.csl.sri.com/~neumann/`). When commercial systems are not adequately robust, we must consider how sound open-source components might be composed into demonstrably robust systems. This requires an international collaborative process, open-ended, long-term, far-sighted, somewhat altruistic, incremental, and with diverse participants from different disciplines and past experiences. It also requires serious attention to the reasons why composition has been so risky in the past (as discussed in the debate [145] noted at the end of Section 5.7). Pervasive adherence to good development practice is also necessary (which suggests better teaching as well). The process needs some discipline, in order to avoid rampant proliferation of incompatible variants. Fortunately, there are already some very substantive efforts to develop, maintain, and support open-source software systems, with significant momentum. If those efforts can succeed in producing demonstrably robust systems, they will also provide an incentive for better commercial systems.

Overall, we need techniques that augment the robustness of less robust components, public-key authentication, cryptographic integrity seals, good cryptography, trustworthy distribution paths, and trustworthy descriptions of the provenance of individual components and who has modified them. We need detailed evaluations of components and the effects of their composition (with interesting opportunities for formal methods). Many problems must be overcome, including defenses against Trojan horses hidden in systems, compilers and evaluation tools, in hardware, source code, and object code – especially when perpetrated by insiders. We need providers who give real support; warranties on systems today are mostly very weak. We need serious incentives including funding for robust open-source efforts. Despite all the challenges, the potential benefits of robust open-source software are worthy of considerable collaborative effort.

Plans for the collaborative research and development of trustworthy survivable (e.g., robust, secure, reliable) interoperable nonproprietary open-source software components are beginning to germinate. We must seek an open process that encourages the development of systems and components addressing the essential problems defined in this report, and which might initially be called Pretty Good Survivability (PGS). The intent is that, through long-term open collaborative efforts involving research and development communities and universities, PGS could gradually evolve into Very Good Survivability (VGS). At the moment, VGS seems like a dream, but it seems to be feasible if PGS is suitably motivated. It also seems absolutely essential to the future of highly survivable systems, and should be well worth whatever effort it requires.

A discussion group for the encouragement of a movement to produce robust nonproprietary software (whether "open-source" or "free") has been formed, and is beginning to have insightful contributions. (To join, send e-mail to `open-source-request@CSL.sri.com` with the one-line content `subscribe` – or `subscribe` [your address] if your desired address is different from your `from:` address; Majordomo will accept contributions only from your specified `to:` address.)

5.10 Integrative Paradigms

Reliability, fault tolerance, security, and indeed survivability must be conceptually integral to hardware and software, despite the desire to use off-the-shelf weakware as the basis for critical applications. In principle, mainstream concepts should be used where applicable, although their shortcomings must be overcome. Good software engineering practice should be used in applications as well as system development. The entire process of program development should be systematized wherever possible. Formal methods should be applied to particularly critical algorithms and programs. Several potentially useful new research directions are also noted below.

As described in Section 4.3, the supercomputing field has suffered in the past from a serious case of myopia. Some of the lessons that can be drawn from that experience are directly applicable to the need for highly survivable systems and networks.

5.11 Fault Tolerance

There is a vast literature of techniques for fault tolerance that this report does not attempt to replicate. For example, techniques for increasing system reliability in response to hardware faults and communications failures are explored in general in [40, 90, 114, 151, 159, 224, 265, 281, 284, 321]. Failure recovery in the context of Tandem's NonStop Clusters is considered by Zabarsky [353], representing a serious step toward systemic fault tolerance. Some significant recent research of Kulkarni and Arora relates to compositionality properties of fault tolerance [21, 149] and the somewhat canonical decomposition of fault-tolerant designs into *detectors* and *correctors* [22].

Once again demonstrating the desirability of a confluence of requirements and a corresponding confluence of techniques for combatting security and reliability problems along the lines of the reconvergence of availability requirements in Figure 3.1, consider the requirements for data integrity in the sense of no-unintended-change shown at the nodes designated by a sharp (#) in the figure. Data integrity can be enhanced through cryptographic integrity checks (typically to protect against malicious alterations) or error-correcting coding techniques (typically to protect against accidental garbling). However, an interesting recent special-purpose use of coding for detecting malicious tampering as well as accidental errors in once-writable optical disks is given by Blaum et al. [41], taking advantage of the asymmetry inherent in certain once-writable storage media in which writing can change the state of a bit only in one direction (e.g., from a not previously written *zero* bit value to a written *one* bit, but never the reverse). This is another example of a crossover implementation that can simultaneously address different sets of subrequirements stemming from otherwise independent-seeming major requirements. In such cases, considerable benefit can be obtained by recognizing the commonality among otherwise independent subrequirements and then providing a unified treatment in the design and implementation.

5.12 Static System Analysis

Many techniques exist for the *a priori* analysis of system behavior, based on consideration of requirements, design specifications, implementation, and operational procedures. These techniques may be formal (see Section 5.8) or informal. Examples of such techniques are

- Analysis of the internal consistency and completeness of requirements
- Analysis of the consistency between design and requirements
- Analysis of the consistency between design and implementation
- Analysis of operational procedures (e.g., configuration control) and their consistency with requirements
- Black-box analysis (with no knowledge of the innards) and white-box testing (with complete knowledge of the innards), whether with testing or formal methods
- Compiler consistency checking and implantation of dynamic checks as a byproduct of compilation
- Risk analysis, risk management, and risk mitigation
- Independent verification and validation
- Personnel certification
- Product certification

5.13 Real-time Analysis of Behavior and Response

As noted in Section 4.1, there is a great need for the ability to provide real-time detection and analysis of system and network behavior, with appropriate real-time responses — from the coordinated perspective of survivability and its subtended requirements. There has been considerable work on this topic for more than a decade.

SRI has pioneered work on rule-based expert system analysis and statistical analysis, through IDES (Intrusion Detection Expert System [177]) and NIDES (Next-Generation IDES [11, 12, 132, 130]). The current work on EMERALD (Event Monitoring Enabling Responses to Anomalous Live Disturbances) [171, 275, 276] is the current extension of IDES and NIDES to monitor network activity. Overall, we know of no efforts other than EMERALD that are oriented toward the ability to detect problems arising in connection with generalized survivability. (See `http://www.csl.sri.com/intrusion.html` .)

Of course, many other institutions have been developing systems addressing various aspects of the intrusion-detection problem, typically using either rule-based techniques or statistical analyses, but in most cases not both, and usually dealing with users of individual systems or local networks. See Edward Amoroso's new book [10] for an introduction to the field. Many papers are worth reading, including [59, 76, 116, 146, 147, 274, 308]. Bradley [58] considers the effects of disruptive routers. In addition, only a few efforts have addressed fault detection in this context — for example [131, 181, 194].

Schneier and Kelsey [314] have developed a cryptographically based step toward the securing of audit logs against tampering and bypassing.

Another form of real-time analysis involves dynamic network management. Network management should also be integrated with real-time anomaly and misuse detection and real-time reconfiguration as a result of detected problems.

5.14 Standards

Standards are important, but can also be extremely counterproductive if poorly conceived or misapplied. Chapter 6 considers the existing and emerging evaluation criteria. Chapter 7 summarizes some of the Department of Defense efforts to standardize architectures and security services. In particular, Section 7.1 considers the attempt to impose standardization through the Joint Technical Architecture (JTA); Section 7.2 considers the DoD Goal Security Architecture (DGSA); Section 7.3 considers the Joint Airborne SIGINT Architecture (JASA) Standards Handbook (JSH).

Criteria for security are considered in Section 6, including the U.S. Department of Defense Trusted Computer Security Evaluation Criteria (TCSEC), the European (ITSEC) and Canadian counterparts (CTCPEC), and the new international Common Criteria.

The British Ministry of Defence has established some rigorous standards for safety-critical systems [338, 339], although it is not clear to what extent they have actually been used.

As an international nongovernmental organization, the Internet Engineering Task Force (IETF) (`http://www.ietf.org`) has been particularly effective in establishing Internet standards, with considerable emphasis on interoperability and change control. (The IETF strongly favors open interfaces, and tolerates proprietary standards only where open standards also exist.) In addition, other standards are also emerging from the Open Group (`http://www.opengroup.org`), the IEEE (`http://www.ieee.org`), the Association for Computing (ACM) (`http://www.acm.org`), and other organizations.

5.15 Research and Development

Historically, research has provided some powerful techniques for increasing survivability, reliability, and security, although much of the potentially most valuable research has not found its way into commercially available personal computer products, and only occasionally into computer systems. Serious research is still needed to address some of the remaining deficiencies.

- Fault tolerance is a field in which there is extensive and diverse research, many aspects of which have found their way into common practice. Successful examples include alternative network routing, redundant data sites, fault detection and recovery, automatic retry in processing and communications, and error-correcting coding in transmission and storage.

- Security research is a more difficult issue. Much past research has been devoted to unrealistic approaches. Work on multilevel security has had little real impact, for a variety of reasons, one of which is that the MLS concept has not been widely accepted outside of DoD — either in the commercial system and networking mainstreams, or by user communities. As a consequence, many DoD systems still run system-high. Similarly, potentially important research on capability-based system architectures and hardware-enforced domain structures has remained largely unimplemented in commercial systems. Even when hardware does provide some domain separation, the operating systems tend not to use it. Perhaps worst of all, some very realistic and practical research has simply not found its way into general practice. For example, excellent research efforts on authentication, access control, and accountability have simply not been adopted in most commercial systems. Recommendations for achieving multilevel security despite the absence of multilevel-secure user systems and for addressing some of the potentially research-motivated system issues are considered in Chapter 8.

- Survivability resembles the tail trying to wag the dog. Perhaps most important in the future will be system-oriented research efforts that pick up the challenges put forth in this report, integrating reliability, fault tolerance, security, performance, and other vital requirements, within the overall context of generalized survivability.

In this report (see Section 8.2), we pursue the notion of generalized multilevel survivability (MLX, introduced in Section 1.2) that draws on past experience with multilevel security, multilevel integrity, and multilevel availability. We do this not with the expectation that system developers will rewrite all their systems, but rather with the expectation that the MLX concept might provide some useful architectural insights.

The mobile-code paradigm is an important topic for future R&D, with respect to security and reliability. (See Section 8.4.)

Research results also suggest some dramatic changes in high-performance computing, which if properly applied could reverse the rather negative historical perspective noted in Section 4.3. For example, two recent software-based efforts are illustrative of a kind of new thinking that could be very beneficial. Each is a different new paradigm that has considerable potential in the development of high-performance systems.

- GLU [129] provides a virtualization of multiprocessing that requires the addition of a few lightweight statements to conventional programs, which when preprocessed into the conventional language (C in the case of the existing GLU implementations) enable the parallelized execution of the program irrespective of the underlying hardware configuration.

- Simple Maude [167] provides a machine-independent programming language based on rewrite rules that can be compiled to execute on arbitrary hardware configurations. It is a subset of Maude [193], which is an extensive environment including a very powerful metalanguage that serves as an executable specification language and a universal description language (`http://maude.csl.sri.com`).

Specific recommendations for future research and development are given in Section 10.2. The R&D recommendations of the President's Commission on Critical Infrastructure Protection are summarized in Section 10.6.

5.16 Education and Training

Issues such as reliability, security, and system survivability need to become a part of a broader educational curriculum and institutional training programs. The same is true of an understanding of vulnerabilities, threats, and risks. The desired audience includes not just programmers and system developers, but also administrators, legislators, system procurement agents, and even prospective users. However, in the final analysis, education and training cannot be effective unless effective system solutions are available to be learned. The appendix outlines a course curriculum for survivability.

5.17 Organizations

Chapter 7 of the report of the President's Commission on Critical Infrastructure Protection [182] recommends the establishment of some new organizational entities. It is worth reviewing them, because they bear directly on the problems of infrastructure survivability.

- An **Office of National Infrastructure Assurance** in the White House to serve as the focal point for infrastructure assurance
- A **National Infrastructure Assurance Council** of corporate leaders, state and government representatives, and Cabinet officers
- An **Infrastructure Assurance Support Office**
- A federal **Lead Agency** for each sector
- A **Sector Infrastructure Assurance Coordinator** for each infrastructure
- An **Information Sharing and Analysis Center** of government and industry representatives
- A **Warning Center**

This seems to represent a considerable increase in the institutionalization of an already highly bureaucratic situation, especially in that the PCCIP has focused largely on the so-called critical national infrastructures and seriously underplayed the importance of the computer-communication infrastructures. Very little in the PCCIP report suggests that the survivability, security, availability, or reliability of the computer-communication and information infrastructures would gain significantly from these organizational entities. In addition, there is still no constituency for the non-DoD non-U.S.-Government user public, as has been pointed out on various occasions — including in the 1990 in the *Computers at Risk* study [67].

In the meanwhile, President Clinton has reconstituted the PCCIP concept by creating a Critical Infrastructure Assurance Office (CIAO), and created the office of the National Coordinator for Security, Infrastructure Protection, and Counter-Terrorism, which will be responsible for a broad range of policies and programs related to cyberterrorism. In addition, the FBI is establishing a National Infrastructure Protection Center (NIPC) to counter individuals and organizations that commit computer crimes. (See Presidential Decision Directives PDD 62 on counterterrorism and PDD 63 [68], aimed at reducing the vulnerabilities.)

Unfortunately, the U.S. Government has had little success in enticing certain major commercial developers to do the right thing — namely, to significantly increase the survivability, security, and reliability of their systems. That shortcoming may ultimately be the limiting factor — despite the hopefulness expressed in some of the recommendations of our report.

Also, unfortunately, the Government seemingly has not had much success in achieving a minimal level of competence in avoiding security risks (as evidenced by Deputy Secretary of Defense John Hamre calling the Cloverdale kids' cookbook attack the "most organized and systematic the Pentagon has seen to date" – see the Risks Forum, volume 19, issue 60 (which is available at `http://catless.ncl.ac.uk/Risks/19.60.html`) or in dealing with computer systems at all (as evidenced by the deplorable inability to surmount the Y2K challenge – see Congressman Stephen Horn's generally condemning Y2K report card, which is updated periodically at `http://www.house.gov/reform/gmit/y2k/index.htm`). It is a huge challenge merely getting competence levels in security up to the levels suggested in Donn Parker's new book, *Fighting Computer Crime* [248].

Chapter 6

Evaluation Criteria

There currently exist no evaluation criteria that are comprehensively suitable for evaluating highly survivable systems and networks. Even with regard to security in isolation, the existing criteria are incomplete and inadequate. In addition, there is almost no experience in evaluating systems having a collection of independent criteria that might contribute to survivability, and the interactions among different criteria subsets are almost unexplored outside of the context of this report. Nevertheless, a good set of security criteria — if it existed — would be very valuable.

This section considers the emerging Common Criteria effort, which is attempting to overcome many of the deficiencies of its precursors, the DoD Trusted Computer Security Evaluation Criteria Rainbow series (e.g., the TCSEC [211], TNI [209], and TDI [210]), the European ITSEC [91], and the Canadian CTCPEC [64].

The evolving Common Criteria document has been undergoing extensive review, preparatory to being submitted as an ISO standard. See `http://csrc.nist.gov/cc` for the latest draft documents and progress toward establishing the Common Criteria. (Version 2.0 was posted 22 May 1998.)

Any set of requirements, and indeed any generic (abstract) systems architecture, must not overly constrain the implementations of systems intended to satisfy those requirements. This is an inherent danger in the TCSEC, but less so in the other criteria because they are frameworks for evaluation rather than prescriptive requirements. In addition, the ITSEC and CTCPEC effectively distinguish functional requirements from assurance requirements, and that useful distinction has been continued in the Common Criteria.

There is also a serious danger of underconstraining the resulting systems and networks. For example, the Rainbow series of trusted-system criteria may overconstrain implementations with respect to the bundling of criteria elements at a particular evaluation level (e.g., A1, B3, B2, B1, C2), but also underconstrain the implementations with respect to many other criteria elements that are omitted — relating to networking, application security, modern authentication (e.g., using one-time tokens instead of fixed reusable passwords), fault tolerance, reliability, real-time performance, interoperability, reusability, software engineering, and the development process, to name just a few. These aspects are absolutely fundamental to the successful procurement and development of suitable systems and networks that can satisfy stringent requirements. Simply adhering to very superficial but allegedly definitive generic requirements and criteria (Orange Book, Red Book, and others), procurement cookbooks, and Chinese menus for system configuration is doomed to failure. In addition, despite the enormous proliferation of the Rainbow series in multitudinous colors, the TCSEC is intrinsically incomplete, for a variety of reasons.[1] For example, it deals primarily with confidentiality in centralized systems (failing to keep up with the last decade of progress in distributed systems and networked systems, and not adequately treating integrity and the prevention of Trojan horses and other pest programs). It is monolithic, in that it lumps together functionality and assurance, and within functionality criteria lumps together requirements that are more rationally treated somewhat independently. For example, the notion of fixed passwords does not make much sense in systems that demand high assurance. Cryptography is basically ignored. The TCSEC does not adequately concern applications and systems configured out of other systems, stressing primarily trusted system components. It also typically ignores survivability, reliability, fault tolerance, performance, interoperability, real-time requirements, system engineering and software engineering, system operations, and many other issues that are essential to the development and configuration of survivable systems and networks.

The desire to be able to configure critical systems out of off-the-shelf components and particularly off-the-shelf software is commendable, but largely a fantasy. Commercially available infrastructure components (operating systems, database management systems, networking software, and application software) are typically not able to fulfill stringent requirements. In some cases extensive customization is required, and is still inadequate. Furthermore, considerable expertise is required to operate and maintain the resulting systems. The concept of turn-key systems satisfying extremely complex critical requirements is unrealistic.

What is needed in the future is more efforts aimed not at cookbooks but rather at constructive documentation of worked examples providing the following:

- Detailed functional requirements encompassing realistically complete sets of requirements relevant to survivability (e.g., security, reliability, and fault tolerance) that could be used for a wide range of procurements

- Detailed guidance on how to handle the interactions among the various constituent subtended requirements for survivability

- Design frameworks and alternative families of logical architectures and architectural structures that facilitate the composition of survivable system and networks out of subsystems

- Guidance on good system engineering and software engineering practice, and far-reaching approaches toward system development

- Guidance on good procurement practice

- Identification of interoperable development environments and constructive tools. However, it must be recognized that tools are not a panacea; their use may in fact be counterproductive.

---

[1] Several critiques by Neumann [226] and Willis Ware [349] are still valid, because the criteria have not changed.

- Guidance on achieving interconnectivity and reusability
- Identification of fundamental gaps in the existing standards, and identification of which standards are or are not truly compatible with which others
- Identification of fundamental components that are absent or inadequate in the existing products, and guidance on what is actually commercially available rather than merely a (possibly) emerging product.
- Guidance on hardware-software tradeoffs, particularly with respect to cryptographic implementations. (There is a great need for hardware-based authentication as in Fortezza, but hardware implementation of cryptographic algorithms is by no means a panacea. For example, see [227] for an analysis of whether crypto implemented in hardware is necessarily more trustworthy than when implemented in software. Hint: "It ain't necessarily so.")

Chapter 7

DoD Attempts at Standardization

> *The more you know, the less you understand.*
> Lao Tze, *Tao Te Ching*

Several efforts have been made to provide some standardization for systems, the first three driven by the organizations under the U.S. Department of Defense, the fourth resulting from a nongovernmental task group that is explicitly targeted at advising the DoD.

7.1 The Joint Technical Architecture

We focus initially on the Army Joint Technical Architecture (JTA), Version 5.0 [84] and the extent to which it is relevant to the development and configuration of systems and networks with stringent survivability requirements. We consider the JTA to be seriously deficient, and are compelled to discuss the reasons therefor.[1]

7.1.1 Goals of JTA Version 5.0

The three main goals of the Army Joint Technical Architecture (JTA) are very worthy: (1) provide a foundation for seamless interoperability among a very wide range of systems; (2) provide guidelines and standards for system development and acquisition that can dramatically reduce cost, development time, and fielding time; and (3) influence the direction of commercial technology development and R&D investment to make it more directly applicable. The intent of our present effort as described in this report is completely in line with those three goals.

To engage in a meaningful evaluation of the JTA, we must refer to the specific definitions of the three types of "architecture" defined therein.

---

[1] If the JTA were to become a total nonissue in the present context, this entire chapter could disappear from the final version of our report. We include it here primarily because our earlier analysis [234] requested by the Army CECOM at the end of 1995 appears to have been largely ignored, and because that analysis is symptomatic of deeper issues that are absolutely fundamental to survivability.

- A *Technical Architecture* (TA, JTA5.0 Section 1.1.2.1) is the "minimal set of rules governing the arrangement, interaction, and interdependence of the parts or elements whose purpose is to ensure that a conformant system satisfies a specified set of requirements. The TA identifies the services, interfaces, standards, and their relationships. It provides the technical guidelines for implementation of systems upon which engineering specifications are based, common building blocks are built, and product lines are developed."

- An *Operational Architecture* (OA, JTA5.0 Section 1.1.2.2) "is a description (often graphical) of the operational elements, assigned tasks, and information flows required ... It defines the type of information, the frequency of exchange, and what tasks are supported by these information exchanges."

- A *Systems Architecture* (SA, JTA5.0 Section 1.1.2.3) is "a description, including graphics ['graphical representations' is presumably intended], of the systems and interconnections ... . The SA defines the physical connection, location, and identification of the key nodes, circuits, networks, ... platforms, etc., and allocates system and component performance parameters. It is constructed to satisfy [the] Operational Architecture per standards defined in the Technical Architecture. The SA shows how multiple systems within a domain or an operational scenario link and interoperate, and may describe the internal construction or operations of particular systems in the SA."

The ellipses in these quoted definitions denote our elimination of references to the "warfighter" — because the scope of the JTA is explicitly intended to apply "to all systems that produce, use, or exchange information electronically" (JTA5.0, Section 1.1.3), including systems of other Armed Services. In the spirit of trying to use commercial systems wherever possible, it is vital that those systems be adequate for defense purposes rather than requiring extensively customized special-purpose systems.

The concept of the technical architecture must be understood within the overall problem of developing, configuring, and operating compliant systems – a process that is by no means a cookbook type of activity. By itself, the JTA is merely a set of guidelines, with no assurance of completeness or adequacy. Many of the requisite standards are not even established or sufficiently well defined, particularly with respect to survivability, security, reliability, and fault tolerance. Furthermore, highly survivable systems cannot be merely composed out of existing components, as noted in Chapter 4; the existing computer-communication infrastructures are still fundamentally flawed with respect to their ability to address many of the essential requirements. Consequently, our report is considered to be an essential additional set of guidelines, techniques, and principles for the development and procurement of highly survivable systems. We believe that our approach is generally consistent with the intent of all three of the technical, operational, and systems architectures.

7.1.2 Analysis of JTA Version 5.0

The following recommendations are taken almost verbatim from an earlier assessment [234] of various ATA Version 4 drafts, written by Peter Neumann and Peter Boucher of SRI's Computer Science Lab. Those recommendations still seem timely, and in many ways anticipate our present study. We hope that our analysis may become obsolete as a result of subsequent improvements to the JTA that might occur during our project. Further changes to the JTA are increasingly essential, despite the fact that there have been almost no improvements in the JTA (apart from its renaming) in the past two years.

- We recommend that significantly greater attention be paid to a complete, realistic set of system requirements, including not just security, but also survivability, reliability, performance, and other desirable attributes, and that those requirements be systematically integrated into the entire architectural concept. Similarly, those requirements must be integrated completely into any resulting architectures.
- We recommend that significantly greater attention be paid to generic, reusable, abstract, *logical* architectures and architectural structures that can be readily implemented in different platforms, compliant with the JTA.
- We recommend that significantly greater attention be paid to providing guidance on how to effectively procure and develop compliant systems and networks that can demonstrably meet critical requirements.
- We recommend greater effort be devoted to identifying gaps in the existing standards and commercial products, with the long-term goal of inducing vendors to eliminate or reduce those gaps. Efforts to design logical architectures and to provide practical guidance would both help to smoke out some of the remaining gaps.

Ultimately, any system development and its operation depend on (among other things) (1) the availability of an accurate, flexible, realistic, and essentially complete set of functional requirements, (2) the existence of a conceptual architecture that can demonstrably satisfy those requirements, (3) a development process that can tolerate changes to the requirements and architecture during development, (4) competent personnel on the Government side, and (5) development personnel who provide a mixture of abilities including pervasive understanding of the requirements, appropriate technical skills, diligence, conscientiousness, responsibility, and a good practical sense regarding the development process. Without those constituent elements, the notions of a technical architecture, a systems architecture, and an operational architecture are of limited merit.

One of the biggest problems in the past has been that the initial requirements were improperly and incompletely stated and that the effects of subsequent changes could not be properly managed. This problem must be adequately addressed in the very near future.

The JTA Version 5.0 definition of a systems architecture suggests that a systems architecture is a physical implementation. That is in general a very unsound practice. A systems architecture should never exist only as a physical implementation, and is most desirably preceded by a conceptual *logical* architecture. More specifically, a physical implementation represents a system build, not an architecture. That definition violates basic principles of generality, abstraction, and reusability, and puts the cart before the horse. What is needed is a true systems architecture, namely, a logical realization of the functional requirements that can be readily converted into a physical implementation, but that does not overly constrain the physical implementation. There is always a serious danger of trying to use software solutions where hardware is essential, or of using inappropriate hardware where simple software approaches would suffice. A logical architecture must not be locked into irrevocable decisions that a few years later become totally obsolete.

Open architectures are essential. We strongly recommend recognition of the need for logical architectures in which there is considerable emphasis on servers (e.g., file servers, network servers, and authentication servers) and in which trustworthiness can be focused where it is most needed rather than distributed broadly. (See [241] and [280] for examples of how this can be done. Also, see [227] for a discussion of what trustworthiness can be accomplished if the cryptography is implemented in software instead of hardware, and some of the pitfalls of trying to rely on inadequately trustworthy infrastructure.)

Not sufficiently evident in the suite of three types of "architectures" is the notion of a generic, system-implementation-independent, reusable, abstract, architectural structure that could be implemented on different platforms to satisfy possibly related but different requirements, without prematurely constraining design decisions. Although in some sense an intent of the JTA approach, it is not sufficiently well motivated within the framework of the three JTA "architectures".

7.1.3 JTA5.0 Section 6, Information Security

Chapter 6 of the JTA Version 5.0 is largely the same chapter that was rather belatedly incorporated into the ATA Version 4 to address information security, and is relatively unchanged despite the passage of two years — although references have been added to the more recent DoD Goal Security Architecture (DGSA) intended for use with future systems. (See Section 7.2 for discussion of the DGSA.) Unfortunately, the JTA has not been upgraded to adequately address its earlier deficiencies, and almost completely ignores survivability issues and survivability-related requirements other than security. This seems seriously shortsighted.

The following comments are essentially the same as those registered in January 1996 with respect to the drafts of ATA Version 4. It appears that only a few of our earlier comments were actually addressed, and thus the relevant comments from [234] are repeated here — updated to encompass survivability issues.

Much greater guidance on how to interpret these standards and protocols needs to be included. Such guidance is absolutely essential to making the JTA approach practical.

It is still not clear how survivability policy, threats, vulnerabilities, and acceptable risks are intended to fit into the three "architectures". Presumably they are beyond the scope of the JTA, and must be addressed in the operational and systems architectures. But it seems to be impossible for the JTA to fulfill its intended purpose unless those concepts are explicitly accounted for. The process of establishing and accommodating the policy, threats, vulnerabilities, and acceptable risks must somehow be made explicit.

Some additional pointers to relevant standards and guidelines are nevertheless needed here. For example, a reference to NCSC-TG-007 ("A Guide to Understanding Design Documentation in Trusted Systems," Version-1, 2 October 1988, or any successor) might be useful for policy. CSC-STD-003-85 ("Guidance for Applying the Trusted Computer System Evaluation Criteria in Specific Environments" — the Yellow Book) provides useful guidance for applying the NCSC security criteria, as does AR 380-19, Appendix B. Procedural, personnel, and physical security are also considered in AR 380-19. However, AR 380-19 is a bare-bones minimum, and itself is too hidebound by the TCSEC. For example, the password policy is somewhere out of the dark ages, and reflects none of the risks of passwords passing over an unencrypted network or otherwise vulnerable to capture, irrespective of how they are created. Relevant COMSEC policy standards would also be relevant here. Risk standards are mentioned only in passing, but should be referenced explicitly.

In JTA5.0 Section 6.2.1.1, the Orange Book (5200.28-STD) is mandated. (It is not even the best thing available, but at least some of its serious flaws and shortcomings should be recognized.) It would seem to be appropriate to mention the Yellow Book (CSC-STD-003-85, noted above) and at least acknowledge its limitations as well. Unfortunately, this is an example of why the mere mention of such references is inadequate — we are not dealing with a cookbook process. Also, what about NCSC-TG-009 (Computer Security Subsystem Interpretation)? What about the Common Criteria as an emerging standard, which when instantiated with at least Orange-Book-equivalent requirements represents a significant improvement [although it still leaves much room for incompletely specified criteria]?

In JTA5.0 Section 6.2.2.1, the Fortezza material should be revisited.

In JTA5.0 Section 6.3.1.5, given the controversy and confusion relating to the Trusted Network Interpretation, it is not clear what is mandated. In addition, the TNI Interpretation document (NCSC-TG-011) should be mentioned.

Although it has been improved a little since the Version 4.x drafts, Table 6-1 (Protocols and Security Standards) could still become a much more useful table. The left-hand column still amorphously lumps together application, presentation, and session layers, transport and network layers, and in a second grouping datalink and physical layers. The middle column lists a few rather generic protocols. The right-hand column lists rather haphazardly a collection of security-related standards and protocols, with no relation to the middle column, whether the standards and protocols are actually adequate for any intended purposes (and if so, which), and no indication of whether any meaningfully secure implementations actually exist. (For many of these, the existing protocols and their implementations are seriously flawed.)

It would be useful to have a separate table (updated regularly) providing some guidance as to the state of the art, and enumerating current or anticipated implementations that are relevant. This table does not seem to fall naturally into any of the three "architectures", too specific for the JTA, too general for the systems architecture, and not appropriate for the operational architecture. The need for it suggests a new class of "architecture" documents, perhaps somewhat akin to the "interpretation" documents within the TCSEC rainbow series, such as the TCSEC [211], TNI [209], and TDI [210] and the corresponding interpretation documents for the TCSEC [212] and TNI [208]. Such an interpretation document could also include guidance on how to go from requirements ("operational architecture") to a generic architecture to the systems architecture, compliant with the JTA, and might also include some guidelines on system and software engineering. A guidance document would be extremely valuable, whether it is a part of the JTA document or otherwise.

JTA5.0 Section 6.4 is still very weak in comparison with the rest of the document. The fact that no standards are yet mandated is not encouraging for those relying solely on the JTA for their understanding of how to proceed.

JTA5.0 Section 6.5 is also weak. For example, much greater guidance is necessary relating to the security, complexity, and usefulness of user interfaces. In general, sophisticated user interfaces are very complex, difficult to use, and full of security flaws. This entire section has not been upgraded in the past two years, despite significant changes. The emerging standards for personal authentication in Section 6.5.2.2 are unrealistic. Zero-knowledge schemes are the tip of a very large iceberg for personal authentication, and everything else relating to cryptographically based one-time token authentication schemes, biometrics, and other approaches is lost in the shuffle by mentioning only Zero Knowledge. Incidentally, this is an area in which the implicit inclusion of the basic TCSEC [211] (DoD 5200.28-STD) as a mandatory standard is misleading, because that document gives the distinct impression that fixed reusable passwords are perfectly adequate! Fixed passwords are in general a disaster waiting to happen, especially in highly distributed environments with components of unknown trustworthiness.

7.1.4 Augmenting the Army Architecture Concept

To give some examples of what is needed to bridge the gaps among the technical architectures of JTA Version 5.0, operational architectures, and systems architectures, several steps need to be taken.

- The requirements process must address the full gamut of relevant requirements, including not just security, but also overall system survivability, reliability, performance, and other desirable attributes. In general, it can be extremely difficult to retrofit any of these attributes into a system that was procured or specified in the absence of sufficient knowledge of the full set of requirements. (However, see [73] for a contrary view.) Thus, much greater guidance is required in facilitating the establishment of a compatible and realizable set of requirements. The existing concept of the operational architecture presumably must include a sufficiently complete representation of the necessary requirements.
- The concept of a generic, or abstract, architecture is completely missing from the outlined process, falling somewhere between the operational architecture (a set of requirements) and the systems architecture (which seems closer to a specific implementation).

In this section and in Chapter 6 on criteria, we paint a rather negative picture of the difficulties that have arisen repeatedly in procuring, developing, operating, using, and maintaining a wide variety of possibly unrelated systems and their networked interconnections. (See [228] for an elaboration and analysis of some of the risks involved.) On the other hand, the Technical Architecture document does represent a possibly useful step toward that goal if supplemented with other approaches. It must not be seen as a magic carpet on which we can fly into the future. We believe that the JTA and indeed the concept of the three Army "architectures" could be much more effective if the contents of our report are taken seriously.

7.2 The DoD Goal Security Architecture

The DoD Goal Security Architecture (DGSA), Volume 6 of the Technical Architecture Framework for Information Management (TAFIM) is intended to represent abstract and generic system architectures. The DGSA and the DGSA Transition Plan for bringing the

DGSA into reality are both considered to be living documents, and therefore the reader is encouraged to check the on-line sources for the latest version (`http://www.itsi.disa.mil/`). Unfortunately, the evolution of these documents has been relatively slow.

The DGSA recognizes, among other things, (1) the importance of different security policies, (2) the reality that users and resources may have different security attributes, (3) the essential nature of distributed processing and networking, and (4) the need for communications to take place over potentially untrustworthy networks. It may be a useful step forward, but the proof of the pudding is in the eating and its potential utility is not yet realized.

A recent examination of the DGSA is given by Feustel and Mayfield [97], who remark that "Perhaps the DGSA document can best be viewed as a conceptual framework for discussing a security policy and its implementation." However, the DGSA leaves completely open which security policies are to be invoked. See also Lowman and Mosier [173], investigating the use of DGSA as a possible methodology for system development — and giving two illustrative systems so described.

According to the DGSA document, an *abstract architecture* grows out of the requirements; it defines principles and fundamental concepts, and functions that satisfy those requirements. A *generic architecture* adds specificity to the components and security mechanisms. A *logical architecture* applies a generic architecture to a set of hypothetical requirements. Finally, a *specific architecture* applies the logical architecture to real requirements, fleshes out the design to enable implementation, and addresses specific components, interfaces, standards, performance, and cost.

For discussion purposes, the overview of the TAFIM volumes gives this summary of the TAFIM Volume 6: The DGSA "addresses security requirements commonly found within DoD organizations' missions or derived as a result of examining mission threats. Further, the DGSA provides a general statement about a common collection of security services and mechanisms that an information system might offer through its generic components. The DGSA also specifies principles, concepts, functions, and services that target security capabilities to guide system architects in developing their specific architectures. The generic security architecture provides an initial allocation of security services and functions and begins to define the types of components and security mechanisms that are available to implement security services. In addition, examples are provided of how to use the DGSA in developing mission-level technical architectures."

Although security policy is left unspecified in the DGSA, the requirements are focused on security and do not address other important aspects of survivability. In principle, that should not be an obstacle if the DGSA is sufficiently general to encompass survivability requirements as well. The extent to which this will be possible remains to be seen. In any case, in Chapter 8 we address architectural structures that could enhance the realization of the generalized survivability requirements outlined in Chapter 3.

7.3 Joint Airborne SIGINT Architecture

The Joint Airborne SIGINT Architecture (JASA) Standards Handbook [112] (JSH) includes Chapter 7 (Security Services) and Annex 10 (Information Security and Information Systems Security Engineering). The handbook's Chapter 7 and Annex 10 appear to carry on the cookbook characteristics of the JTA and DGSA; they attempt to elaborate on the JTA and the Unified Cryptologic Architecture (UCA), which summarizes the primary cryptographic standards. Although such an enumeration of standards is certainly necessary, it leaves much unspecified and does not address the issue of compatibility; in particular, it leaves as an exercise to the reader how to achieve compatibility among incompatible approaches. It also exhibits a remarkable fascination with a secure single sign-on (SSSO) despite the reality that many of the components along the way typically cannot be trusted. This seems to be a monstrous oversimplification, and is dangerous because it is likely to encourage simplistic solutions that are not adequately secure. The handbook does recognize the need for SSSO approaches that can overcome the difficulties arising from multiple authentications (e.g., having to remember different account IDs and passwords), but the risks of trusting untrustworthy components and the serious risks of fixed passwords themselves are not adequately considered. (See Section 7.3.3.4.2 of the 30 June 1998 draft of the JASA Standards Handbook, Version 3.0.)

7.4 An Open-Systems Process for DoD

The final study report of the Open Systems Task Force (OSTF) was scheduled for release on 8 September 1998 [244], but as of the date of this report is for reasons unknown still not available for public distribution. The study group consists of a distinguished group of nongovernment people, sponsored by the Defense Science Board. The report is concerned primarily with the role that open systems might play in Department of Defense computer-communication systems. The report focuses on the requirements of an open-systems process for DoD, and characterizes the advantages of this approach.

A very compelling statement is included in the Executive Summary of the OSTF Version 7.9 draft of 21 August 1998: "The Task Force argues that major DoD priorities *cannot be achieved* without a massive infusion of Open System attributes through an organized Open Systems Process. Some sort of Open Systems Process must become the DoD *mindset* and *core value*."

We make a distinction here between the open-system concept (which implies primarily that heterogeneously different systems can in some sense interoperate) and the open-source concept discussed in Section 5.9 (which *additionally* implies that the source code and interfaces are publicly available in some meaningful sense). Many of the arguments for the open-system approach recommended by the OSTF report are also relevant as motivation for pursuing the intrinsically open-source approach recommended in this report. However, the open-source approach is in our opinion much more compelling and much more far-reaching. In any event, we strongly urge the DoD to encourage both efforts.

Chapter 8

Architectures for Survivability

The appropriate use of structure is still a creative task, and is,
in our opinion, a central factor in any system designer's responsibility.
Jim Horning and Brian Randell, 1973 [124]
(Note: It was already a very creative task in Multics in 1965,
and is still a vital creative task in 1999. It is also a
detailed engineering task. PGN)

The emphasis in this report is on *architectural structures* and *structural architectures* that are independent of particular system and network designs and independent of specific implementations, but still firmly rooted in the broad set of requirements for survivability. In this way, we avoid getting mired in the distinctions among the Joint Technical Architecture's "technical architectures," "operational architectures," and "systems architectures" (Chapter 7.1) (which lack a true sense of architecture) and the DGSA's abstract, generic, specific, and logical architectures, and the so-called security architecture (Chapter 7.2).[1]

Some of the architectural structures considered here involve relatively untrusted end-user systems combined with ultradependable trustworthy servers out of which structural architectures can be conceived, and from which survivable systems and networks can be developed or configured. Of particular interest are architectural structures that include authentication servers, file servers, and network servers, which under generalized dependence can provide highly survivable and highly secure overall systems and networks.

Some of the less-survivable proposed architectures can be achieved by gradual evolution. Unfortunately, some of the longer-term approaches that could achieve truly high survivability require more revolutionary new directions; they are much more farsighted, but less likely to win popular support among system developers bent on lowest-possible-cost solutions.

This chapter considers multilevel-secure systems as well as single-level systems. Single-level systems are ubiquitous. Multilevel-secure systems are desired by the Department of Defense, but introduce many problems of their own — some of which may interfere with the needs for survivability. Ideally, multilevel-secure systems should be configurable with only minimal dependence on multilevel-secure components, rather than requiring pervasive high-assurance MLS throughout every subsystem. Furthermore, the single-level systems should be integrally related to the multilevel systems, rather than completely different families of architectures. If the architecture is properly conceived, a multilevel system should not have to be significantly different from its single-level counterparts. This is a goal that has not previously been pursued, and runs counter to the dictates of the Trusted Computer Security Evaluation Criteria (TCSEC) discussed in Chapter 6. However, it seems highly advisable if MLS systems are ever to become practically achievable. Nevertheless, the inherent incompleteness of MLS requirements must be addressed, in particular with respect to the requirements for integrity and survivability.

---

[1] The last definition of "architecture" in the *Random House Dictionary* is simply "the structure of anything." Although our terms "architectural structure" and "structural architecture" may seem tautologies outside the context of this report, they are intended to emphasize the desired fundamental role of structure, and to avoid confusion resulting from the massive overloading of the word "architecture" that arises in the JTA and DGSA.

8.1 Structural Organizing Principles

Several fundamental architectural principles (e.g., [305]) are particularly relevant to architectural structure, each of which can considerably improve overall survivability. Not surprisingly, they have deep roots in the security and software-engineering communities.

- **Abstraction.** In computer system parlance, abstraction is the ability to describe logical functionality in its own terms rather than invoking implementation-specific details, and to mask such details from the invokers of the given functionality. Abstraction hides implementation details from the purview of the interface. Abstraction is particularly useful in hiding or reducing apparent complexity of a system or subsystem. One important manifestation of abstraction is found in implementation-independent interfaces that can greatly enhance interoperability, reusability, and long-term system evolution.

- **Hierarchical layering.** In most cases, separate layers of abstraction can advantageously be identified and distinguished from one another in terms of the functions performed and the information entities involved. Each abstracted layer should mask most of the idiosyncrasies of lower layers. The functionality at any layer should be specified in terms of the entities at that layer, not in terms of lower-layer functionality. A simple example is given by the layering of Table 3.1, although in a detailed architecture, each of those layers might itself be substructured. Our notion of compromise from outside includes compromises originating from hierarchical abstractions at higher layers as well as from other abstractions at the same layer. For these purposes, we refer to *vertical abstraction* and *horizontal abstraction,* respectively.

- **Encapsulation.** Abstraction masks implementation detail at a particular interface. Encapsulation of the functionality of that abstraction has two additional attributes: it prevents the derivation of internal state information from observation of the interface, and it attempts to prevent compromise from outside — for example, by interference with the internal operations. It is relevant to both design (ensuring that the internals are logically hidden) and implementation (ensuring that the internals are in actuality hidden).

- **The object-oriented paradigm.** The object-oriented paradigm noted in Section 5.6 encompasses the above concepts of abstraction, encapsulation, inheritance, and polymorphism, which collectively provide a potentially valuable structuring organizing principle.

- **Design diversity.** Dependence primarily on one system provider, one software developer, one hardware producer, one set of network software, one set of protocols, one Web browser, or one operating system is inherently unwise if survivability is a serious goal. Common fault modes and common security vulnerabilities can render the entire information infrastucture useless for prolonged periods of time. Thus, diversity is desirable in every one of these dimensions.

- **Composability.** The principle that modules, subsystems, and systems should be readily composable without additional effort is fundamental to the demands of survivability — although the implication is not entirely obvious in a world in which most commercial software is not easily composable. The decoupling of concepts, interfaces, and implementations that is necessary to enhance composability can contribute indirectly to survivability in various ways — for example, by increasing component independence, interoperability, flexibility, generality, and polymorphic usability. The requirement of facile composability offers a strong forcing function on architectures and system configurability, assuming that it is intended to be recognized and honored throughout. Composability can be considerably enhanced by the use of horizontal and vertical abstraction, encapsulation, and separation of policy and mechanism.

- **Design for pervasive authentication and access control.** Although many designers may consider authentication as an afterthought, and access control relevant only for certain access types, security requires a thorough perspective of pervasive authentication and access controls throughout system and network architectures. Uniform treatment of seemingly diverse access mechanisms can be achieved by recognizing common abstractions among different mechanisms.

- **Design for pervasive accountability and recovery.** Similar to authentication and access control, accountability requires a pervasive approach to monitoring of all relevant activities at different layers of abstraction, real-time analysis of situations that might lead to potential crises, and the ability to recover system state information and data when necessary. These capabilities should be addressed in system architecture rather than added on afterwards.

- **Separation of policy and mechanism.** Policy defines what is supposed to be done, and mechanism establishes how it is to be done. Separating them distinctly from one another greatly enhances system flexibility and evolvability, especially if the policies can be changed without having to alter the mechanisms.

- **Assignment of least privilege.** Whenever domain separation and differential access controls can be enforced by a system, permissions should be granted only as needed. Granting of superuser permissions is a flagrant violation of the principle of least privilege.

- **Least common mechanism.** The principle of least common mechanism takes on several manifestations, each of which reflects on not overloading mechanisms with fundamentally different functionality. For example, if you really believe in all-powerful mechanisms (which are inherently dangerous), this principle would suggest the use of a collection of different sub-superuser privileges rather than a single superuser privilege, with different privileges required for different mechanisms. It also suggests that strong typing is preferable to permitting overloading of untyped operations on different types. It further suggests that unrelated functions be treated separately. Although this might seem to run counter to the notion of polymorphism in the object-oriented paradigm (which implies multifunction capabilities), any efforts toward polymorphism should maintain strongly typed usage.

- **Separation of concerns.** We consider *users* at any layer of abstraction as either people or processes that initiate actions in computers at that layer of abstraction. A given user may have different associated duties that are to be performed, and may take on different roles for certain sets of duties. (The notion of "user" may differ from layer to layer. For example, compiled code is the typical user of the hardware. Application programs are often users of operating systems. People are typically users of graphical user interfaces and command tools.) Functional operations may take place in different protection domains. Separation of concerns takes on three corresponding forms.

  - Separation of roles recognizes the existence of different roles (e.g., as a certain kind of administrator or an ordinary user) under which any particular *user* (that is, an individual, or a process, or a TCSEC *subject*) may operate.
  - Separation of duties recognizes that different individuals should operate under different roles according to the duties being performed, and should be allocated permissions accordingly. Separation of roles is necessary for the implementation of separation of duties — that is, the allocation of different access permissions according to the specific role being played.
  - Separation of domains recognizes that different programs or subsystems need to operate in different execution domains, and that separate domains must not be commingled in their design concepts or in their implementation. Separation of domains is also necessary for the implementation of separation of duties.

  The principles of separation of roles and separation of duties are associated with users, whereas separation of domains relates to executable programs. The monolithic Unix superuser mechanism is an example of the inability to distinguish among different roles and different duties, because there is only one superuser domain. (Several systems have recently partitioned the monolithic superuser role according to distinct roles.) Fortran's shared common is another example of the inability to adequately separate domains. System architectures are needed that can effectively enforce rigorous separation among different protection domains, according to different roles. (The definition and composability of separation-of-duty policies is considered in [102].)

- **Avoidance of strict dependence on untrustworthy entities.** The notion of generalized dependence explicit recognizes the risks inherent in depending on something (e.g., a component, system, communications medium, or user) that is either inherently untrustworthy or potentially suspicious. Clearly, strict dependence on such entities should be avoided or otherwise circumscribed, wherever possible.

- **Scrutability of designs and implementations.** Although it seems to be less of a structural organizing principle than a philosophical principle, the concept of scrutable designs and implementations that can be subjected to open peer review is very helpful in gaining confidence and improving confidence in system survivability.

- **Mobile code.** A fundamental potential advantage of mobile code is that the code can be platform independent and executed on any system for which a suitable interpreter exists, irrespective of where the code originated. The Java adage, "*write once, run anywhere,*" is symbolic of that potential. Although the concept of mobile code by itself does not strictly speaking seem to be a structural principle, it cries out for the use of architectural structures in which code and its execution need not coexist. For that reason, it is included in this itemization. Section 8.4 considers the architectural implications of mobile code.

- **Portable computing.** We make a distinction here between mobile code and portable computing. In the former case, the code is mobile. In the latter case, the computing platforms and workstations are themselves portable. The concept of integrating rapidly deployable wireless laptops securely within a highly configurable infrastructure clearly requires a flexible survivable architecture, as well as extensive use of encryption for integrity, confidentiality, prevention of denials of service, and enhancing availability.

For a variety of reasons, these organizing principles can contribute to increased system and network survivability – if they are consistently applied and if they are properly implemented. Note that abstraction, layering, encapsulation, object-oriented approaches, and policy-mechanism separation all can contribute to greater interoperability, reusability, long-term system evolvability, and security. The principles of separation of concerns and least privilege can also substantially improve operational security and reliability.

These principles can also contribute to improved analysis. In particular, formal methods can be used to analyze requirements, specifications, and implementations. However, such analyses can be greatly simplified by the use of structural concepts – especially layering, abstraction, encapsulation, policy-mechanism separation, and domain separation. For example, the mappings among layers of formally specified abstractions in SRI's Hierarchical Development Methodology [292] were capable of inducing enormous simplifications in the formal proof process for large systems.

Approaches that properly address the mobile-code problem demand significant improvements in the information infrastructure. The notion of portable computing is clearly a forcing function on system architectures, and can result in significant improvement of the survivability of the entire system and network complex.

Ideally, modern software engineering should encompass these organizing principles, although in practice it is frequently not used in a sufficiently disciplined manner to take advantage of them.

In direct response to the 1990 *Computers at Risk* report of the National Research Council [67], an effort is proceeding to develop and promulgate a set of Generally Accepted Systems Security Principles (GASSP) (`http://web.mit.edu/security/www/gasspl/html`), and to establish an International Information Security Foundation (I^2SF). Many of those principles are relevant to survivability as well, but are clearly not enough by themselves.

8.2 Architectural Structures

Several main structuring concepts are of particular interest, each of which has the potential of inducing considerable discipline on architectures employing these concepts, and thereby enhancing survivability.

- **Security kernels and TCBs.** Traditional centralized TCSEC architectures have relied heavily on the hypothesis that a relatively small security kernel (and trusted computing base, or TCB) could completely encapsulate the mediation of all security issues. Whereas this is only hypothetically possible for strict-sense multilevel security (and then only incompletely in a networked environment), most other critical security properties cannot be so encapsulated unless the kernel and TCB are large. This approach also runs into difficulties in highly networked environments, in which there can be no single TCB; the notion of a trusted computing base must be either extended to require trustworthiness of all components, or else be able to accommodate systems of potential untrustworthiness. Multilevel security and multilevel integrity controls can help limit adverse effects to within security levels (and compartments) — including malicious and accidental misuse as well as implanted Trojan horses. In addition, Paul Karger [140] shows how discretionary access controls can further limit the potential damage of Trojan horses.

- **Separation kernels.** Backing off from MLS kernels, Rushby [297, 298] proposed a much simpler separation kernel (also called an isolation kernel) that does nothing more than provide domain separation on which resource managers can be built that effectively isolate different users and different uses, and upon which application-layer software can be built. Such a simple architectural structure works very nicely for centralized and nonnetworked applications with elementary isolation requirements. Although in itself it does not support networked systems, the concept of a separation kernel can be extended by having resource managers that work across different system platforms and that virtualize the networking functionality. However, the resource managers must themselves be trustworthy with respect to the networking, as in the case of MLS kernels and TCBs.

- **Isolation mechanisms, such as firewalls and trustworthy guards.** Firewalls, guards, and other isolation mechanisms can increase the trustworthiness of systems that they encapsulate or mediate between, by partitioning systems and networks in such a way that some limited interactions can still be permitted under carefully controlled circumstances. Section 1.2 notes that a basic requirement in mediating between regions of potentially unequal trustworthiness is to ensure that sensitive information does not leak out and that Trojan horses and other harmful effects do not sneak in. Extending that to survivability, the isolation mechanisms must ensure that actions in one region cannot compromise the survivability of another region across the isolation boundary.

- **Read-only bootstraps and backups.** Considerable survivability can be achieved if systems are booted or rebooted from nonalterable sources such as ROM or CDs. This provides an alternative to cryptographic integrity checks that detect unexpected system alterations. There is also a role for once-writable memory media, for audit trails and logs, and for backups that can permit rapid recovery (as in the Plan-9 directory structure, which allows virtual access to a directory structure as of any specified time). (See [314] for a cryptographic alternative.)

- **Multilevel-security and multilevel-integrity TCBs.** MLS and MLI (See Section 1.2.7) each force structure on the resulting systems and networks, particularly if any real assurance is desired (as in TCSEC B3 systems). For example, the design of a general-purpose MLS system demands the hierarchical layering of the system and the complete encapsulation of its security kernel and TCB. It also requires strict separation of domains across MLS boundaries. The use of an MLS system requires explicit separation of roles and duties.

- **Multilevel survivability TCBs (generalized MLX).** The MLX concept defined in Section 1.2.8 with respect to generalized dependence appears to be a powerful potential structural aid in designing and configuring systems and networks with highly critical survivability requirements. As noted there, the use of generalized dependence rather than the strict depends upon relation avoids some of the classical problems associated with the strict partial orderings inherent in MLI. It also encourages the development of trustworthy mechanisms that are implemented out of less trustworthy components, as illustrated by various examples given in Section 1.2.5. This concept is pursued further in Section 8.5.2.

- **Limited-trust architectures.** An architectural concept for multilevel-survivable systems that is of considerable interest here involves largely off-the-shelf multiple single-level (MSL) security in end-user systems whose interconnections with trustworthy servers provide multilevel security and multilevel survivability, generalizing the architectural approach described previously by Proctor and Neumann [280] at SRI, based in part on an earlier Newcastle Connection Distributed Secure System architecture of Rushby and Randell [302] at Newcastle that avoids some of the problems associated with monolithic multilevel-security kernels and TCBs. The challenge is to allow trustworthy systems and networks to be developed or configured without having to trust constituent components that would normally have to be trustworthy under conventional architectures. Examples of generalized dependence in which this is possible are given in Section 1.2.5. This approach is broadened in this chapter. An alternative approach is described by Kang, Froscher, and Moskowitz [138] at the Naval Research Laboratory, and is also considered. The NRL work aims at what is in effect a highly distributed MLS TCB incorporating a variety of efforts to integrate MSL systems, including a one-way-flow architecture (e.g., [80], a Pump [139] and SINTRA on the server side, and two COTS-based switched workstations on the client side, allowing them to operate at different multilevel-security levels). One of the clients is the Starlight Interactive Link[13], developed by the Australian DSTO. The other is the COSPO (Community Open Source Program Office) Switched Workstation, developed by MITRE, and approved for use between Unclassified and TopSecret levels of classification. The latter scheme makes extensive use of integrity checks and authentication, particularly across MLS boundaries. However, neither the NRL work nor Starlight addresses the integrity problems that accompany upward flow from a lower to a higher MLS level — the MLS-only view allows rampant acquisition of Trojan horses and other pest programs.

- **Explicitly compensating system structures.** The Newcastle Recovery Block mechanism [16, 17, 125] represents a rather different approach to system structuring, in which alternative programs are automatically invoked in a systematic way whenever the primary programs do not dynamically satisfy their run-time requirements. In that approach, alternatives can be anticipated naturally within the overall system structure to accommodate arbitrary failures or attacks, resulting in acceptable overall system behavior — for example, fail-safe, fail-soft, fail-hard, fail-secure, fail-unsecure, fail-with-minimal-availability, or in the context of our present goals, fail-survivable, with whatever definition of degraded survivability may be acceptable under the given circumstances.

- **Minimum essential information infrastructures.** The concept of a *minimum essential information infrastructure* (MEII, [15]) has been proposed in the critical infrastructures community to ensure that some essential functionality would continue to exist despite a reasonable range of survivability threats — attacks, outages, failures, environmental disturbances, and so on. Unfortunately, there is considerable debate over the definitions of *minimal* and *essential* — as well as recognition of the problems inherent in trying to focus on a single universal MEII. Because of widely varying requirements (especially when survivability is introduced), it is very unlikely that a single MEII could be acceptable. However, the concept remains valuable in a vague sense as a guiding architectural concept. Furthermore, conventionally based MEIIs are likely to stumble seriously in the presence of subsystems with questionable (minimal?) trustworthiness that are endowed with maximal (blind) trust. Although MEIIs and minimum-trust architectures are not necessarily incompatible in principle, they may be incompatible in practice — given the inherent untrustworthiness of so many of the available commercial system components and the doubts about improvements in the future. A discussion of some of the alternatives and problems with such an approach is given in [312]. We do not repeat them here, because the MEII concept is more of a buzzword than a reality, in light of the wide diversity of requirements subtended by survivability.

- **Classical control theory.** The well-established field of control systems offers various techniques that can be used to increase the survivability achievable in systems, including stability of feedback. See [332] for the use of certain concepts from control theory in developing survivable information systems. Such concepts could in principle add significantly to the architectural structure of a survivable system.

What is likely to be necessary in the long run is the establishment of a family of logical system architectures encompassing the best aspects of those approaches that are really applicable to survivability. For example, we can conceive of systems whose architecture is based on minimizing trustworthiness where possible, using MLS kernels and TCBs in MLS servers where multilevel functionality is essential, using stringent domain separation where multiple users are necessary (but not necessarily in one-user personal computers or in dedicated workstations), using dynamic loading of authenticated mobile code from trusted sites, and using explicitly compensating system structures where that approach can have high payoffs. Such an architecture might actually achieve the desired effects of robust MEIIs; however, the goal of achieving MEIIs is derivative; it would be the result of having developed suitable system architectures, and is not meaningfully achievable by itself.

Multics, PSOS, SeaView, and EMERALD (EMERALD is discussed in Section 5.13) are excellent examples of the role of design structure, because developers of each of those systems took great pains to advance the state of the art in constructive structure and good software engineering practice. (See the citations of Noteworthy References just preceding the bibliography at the end of this report.)

8.3 Architectural Components

8.3.1 Secure Operating Systems

The vast majority of commercial personal-computer operating systems (notably, those from Microsoft) are a joke when considered with respect to network security and availability. Some of the Unix platforms have matured to the point at which early jokes about "Unix security" being an oxymoron are a less serious concern, although the ability to misconfigure Unix systems is still a problem.

In conventional centralized multilevel-secure systems, it is customary to talk about the scope of the security perimeter that encompasses the enforcement of multilevel security — typically a multilevel-security kernel plus some (often large) amount of trusted code in the TCB. However, such a security perimeter does not encapsulate the security concerns, only a selected few abstracted issues relating to multilevel security. As soon as we consider distributed systems and highly networked environments, the so-called security perimeter typically encompasses major components and functionality (such as compilers, run-time libraries, browsers, bytecode interpreters, servers, and untrustworthy remote sites), and in some cases may actually be essentially unboundable — especially when it includes the entire Internet, every telephone in the world, and electromagnetic interference from unanticipated sources.

In all such systems — whether centralized or distributed — with any generality of purpose, there is no survivability perimeter in the sense that all critical survivability issues can be circumscribed. Nevertheless, several of the structural architectures considered in Section 8.5 are capable of providing survivable systems and networks in the absence of secure operating systems for end-user systems. However, authentication becomes a very critical issue, as does the need for trustworthy trusted paths.

8.3.2 Networking Software

We need much better protocols — more robust, more secure, more highly available, and so on — with dramatic improvements over existing ones (TCP/IP v6, ftp, telnet, udp, smtp) that are soundly implemented. Robust networking protocols must also be embedded in sound operating systems; otherwise, they are compromisible — from outside, from within, and from below. It is conceivable that some wrapper technology could provide some short-term help, but given the dramatic increases in bandwidth, it is clear that new protocols are needed anyway. The needs of survivability need to be recognized in new protocol efforts.

8.3.3 Encryption and Key Management

Strong cryptographic algorithms and their robust implementation are absolutely fundamental to the attainment of system security and survivability. Shared-key cryptography (also called secret-key cryptography) is helpful (but not by itself sufficient) in achieving confidentiality, integrity, some detection of denials of service, and in preventing various forms of computer misuse. Public-key cryptography is particularly well suited for key management (key agreement, key distribution), integrity, and authentication.

Unfortunately, even the best cryptographic algorithms can often be trivially compromised from outside, from within, and from below, in a variety of ways. For example, systems that employ key-recovery and key-escrow techniques have intrinsic trapdoors and are likely to be subject to compromise of one form or another — by trusted insiders, but also potentially by outsiders. Hardware-implemented cryptography is often considered to be more secure than software-implemented cryptography, but that is not necessarily the case. (For example, see [227].)

The Diffie-Hellman and Rivest-Shamir-Adleman (RSA) algorithms are extremely important examples of public-key algorithms. (For background, see Schneier's *Applied Cryptography* [313].)

Key management is a very serious concern. As one example of a desirable approach, the Diffie-Hellman public-key technique [86] provides an elegant means for key agreement without the actual key ever having to be transmitted. Agreement is reached with each party using its own private key and the other party's or parties' public key, based on partial information shared among the parties from which each can construct the desired shared key for subsequent secret-key communications.

Only through careful and comprehensive study of vulnerabilities such as those noted in Section 4.1 (e.g., see [6, 7, 14, 79, 148, 313]) is it possible to develop algorithms, protocols, and implementations that are significantly less vulnerable to attack and misuse. Perhaps here more than in any other area of security, the ultimate truth is that there are no easy answers when it comes to the robustness of cryptographic applications.

A broad range of standard specifications for public-key crypto [128] is currently being defined under IEEE auspices. It encompasses public-key cryptography that depends on discrete logarithms, elliptic curves, and integer factorization. In its present advanced draft form, it already appears to be an extraordinarily useful document, and could go a long way toward unifying the cryptographic product marketplace.

8.3.4 Authentication Subsystems

One of the most important subsystems that is not easily attainable today in commercially available systems involves a set of highly survivable trustworthy distributed authentication mechanisms that can support a variety of authentication policies, providing nonspoofable authentication despite the presence of potentially untrustworthy components — such as end-user terminals and workstations, Web servers, intermediate network nodes, and possibly flawed embeddings of cryptographic algorithms. We attempt to characterize some Byzantine-like authentication servers that can operate securely despite such uncertainties, and examine some of the more realistic variants. Thus, there must be multiple authentication servers for higher availability, internal redundancy and cross-checking for reliability, and extensive use of cryptography for confidentiality, integrity, and nonspoofability. An important proposal for a public-key certificate-based Simple Distributed Security Infrastructure (SDSI) is given by Rivest and Lampson [291] along with some sort of Secure Public Key Infrastructure (SPKI). See also Abadi's formalization of SDSI's linked local name spaces [1]. There is a long history of work on systemic authentication, going back to Needham and Schroeder [216] beginning in 1978 (with discovery of flaws, fixes, and other advances since then [172, 200]), MIT's Kerberos [199, 327, 35, 217] beginning around 1987, and the Digital Distributed System Security Architecture (DDSSA) [101, 155] around 1990. SDSI is an outgrowth of that particular chain of intellectual history from the research community. Somewhat independent work stems from the European work on the SESAME project [249].

8.3.5 Trusted Paths and Resource Integrity

An absolutely critical weak link that must be overcome is the absence of a trusted path from the user to the system, particularly in personal computers but also in workstations. Recent work at the University of Pennsylvania by Arbaugh et al. [18] based on their earlier work on the AEGIS Secure Bootstrap [19] presents a new approach that enforces a static integrity property on the firmware and a combination of induction, digital signatures, and modifications to the control transitions from certain major modules such as `call` and `jump` instructions. This approach is called *Chaining Layered Integrity Checks* (CLIC). (See also related work on trusted automated recovery [20], which shares many of the same problems with the trusted path.)

Closely related to, and in some sense a generalization of, the trusted-path problem is the need for assurance that any resource (data, source code, object code, firmware, and hardware) has not been tampered with or otherwise altered. This problem exists whether we are concerned with firmware in local systems, sensitive (that is, survivability, security, or reliability relevant) components of operating systems, middleware, application code, and — very critically in networked Web environments — applets or other software that comes from external sources. It is also very important in backup and retrieval, and in reconfiguration. Essentially any out-of-band change to the system or network state is vulnerable to compromise. Workable approaches may use a combination of digital signatures, cryptographic integrity protection, dedicated tamperproof hardware (particularly for cryptographic functions), proof-carrying code, and other forms of dynamic code checking.

8.3.6 File Servers

Given appropriate uses of cryptography (Section 8.3.3), systems can be designed in which file servers need not be trustworthy with respect to confidentiality or integrity, although there would still be reliability problems relating to guaranteed availability and security problems relating to preventing denials of service and ensuring that the accessed file servers are authentic. This is true even with multilevel security (e.g., [280]). However, given the possibility of a file server being compromised from within or from below, it is usually desirable to ensure that some minimal trustworthiness is provided by the file servers themselves.

8.3.7 Network Servers

Given appropriate uses of cryptography, new network protocols or assiduous overlays on the existing protocols, and careful implementation on relatively secure platforms, it is in principle possible to develop network servers that can be adequately trustworthy. Multilevel security requires either extraordinarily trustworthy operating systems on which to mount the network servers, or else multiple single-level servers (e.g., [280]). Network servers must be designed to provide confidentiality, integrity, protection against denials of service, and fault tolerance.

8.3.8 Name Servers

Although name servers are rather naïvely often thought not to be security critical, they are certainly critical with respect to preventing accidental and intentional denials of service and to achieving overall system and network survivability. Inaccessibility of or attacks on system and network name servers can have devastating effects. Correctness of data is also a serious problem. Name servers can also be instrumental in attacks that use inferences that can be drawn from the information they provide.

8.3.9 Monitoring

Section 5.13 suggests the need for real-time analysis of system and network behavior and appropriate timely responses. As an example of how this might be achieved, we believe that our existing EMERALD (Event Monitoring Enabling Responses to Anomalous Live Disturbances) environment [171, 275, 276] can be readily generalized to address detection, analysis, and response with respect to significant departures from expected survivability-relevant characteristics, including reliability, fault tolerance, and availability in addition to the current emphasis on security. The EMERALD statistical component recognizes anomalous departures from expected normal behavior, whereas the EMERALD signature-based component is rule based and recognizes the presence of potential exploitations of known or suspected vulnerabilities. The EMERALD resolver passes its results on to higher-layer instances of EMERALD running with greater scope of awareness; in a generalized version, it could also mediate conflicts that might arise among the different subtended survivability requirements. We are contemplating integrating EMERALD with a classical network management facility, which would provide real-time information relating to configuration management, performance management, fault management, security management, and accounting management.

EMERALD is at present oriented primarily toward detection, analysis, and response related specifically to security misuse of computer networks. Its basic architecture observes good software engineering practice, abstraction, and internal interoperability, and is naturally well suited to this generalization effort — which we believe will fill a major gap in attaining flexible system architectures for survivability. EMERALD is also participating in the Common Intrusion Detection Format (CIDF) effort, which will enable considerable interoperability among different analysis systems and reusability of individual components in other environments.

We refer to EMERALD here primarily to illustrate the potential applicability of real-time monitoring and analysis in the maintenance of survivable systems. EMERALD has the oldest ancestry and the greatest generality of approach (following its predecessors IDES and NIDES). (See `http://www.csl.sri.com/intrusion.html` for extensive background on our work in this area.)

8.3.10 Architectural Challenges

With reference to the systemic inadequacies outlined in Chapter 4, Table 8.1 summarizes the major hurdles that must be addressed for each of the components of the previous subsections. The table considers the integrity, confidentiality, availability, and reliability of various functional entities, as illustrative of the challenges. It suggests that we still have quite a long way to go.

In light of the extensive set of limitations in the present technology exhibited in Table 8.1 (and the table gives only a sampling), we reiterate that survivability and its subtended requirements of security and reliability are weak-link problems. The table very simply conveys the message that weak links abound. The real challenge is to overcome these limitations.

8.4 The Mobile-Code Architectural Paradigm

A significant new paradigm for controlled execution involves the use of mobile code — that is, code that can be executed independently of where it is stored.[2] The most common case involves portable reusable code that is acquired from some particular sources (remote or local) and executed locally. From a different perspective, it could involve local code that is executed elsewhere, or remote code that is executed at another remote site. Ideally, mobile code should be platform independent, and capable of running anywhere irrespective of how and from where it was obtained. Used in connection with the principles of separation of domains and allocation of least privilege, dynamic linking, and dynamic loading with

---

[2] In 1971, Neumann [222] suggested a framework in which execution could take place with the assemblage of inputs, programs, state information, and outputs, all possibly from a multiplicity of different sites.

Table 8.1: Typical Architectural Limitations

Functionality	**Integrity**	**Confidentiality**	**Availability**	**Reliability**
User PC/NC OS, application code, browsers	Weak security, especially when networked	Weak security, especially when networked	Many crashes, file errors, upgrade woes	HW/SW reliability often poor
Networking and protocols	Flawed designs and code	Flawed designs and code	Weak protocols, code bugs	Weak protocols, code bugs
Cryptography (for integrity, authentication, encryption)	Poor embedding, compromise from within/below; gov't policies	Poor embedding, key exposures, bypasses; gov't policies	Mass market hindered by government crypto policies	Subject to bit errors, key unavailability, synch problems
Authentication subsystems	Reliance on fixed reusable passwords	Reliance on fixed reusable passwords	An outage can shut down all dependent users	Inconsistency among multiple authenticators
Trusted path and resource integrity	Generally nonexistent or very weak	Generally nonexistent or very weak	Denials of service problematic	Generally nonexistent or very weak
Servers (file, ftp, http, e-mail)	Weak security; lacking crypto, authentication	Weak security; lacking crypto, authentication	Outages and service denials, incompatibilities	HW/SW reliability often poor
Monitoring, analysis, response	Bypassable, alterable, spoofable	Sensitive data may be exposed	Not robust, subject to service denials	Algorithms typically incomplete

persistent access controls, this paradigm provides an opportunity for the secure execution of mobile code, and represents a very promising approach for achieving ultrasurvivable systems.

Of course, you can have major integrity, confidentiality, availability, denial-of-service, and general survivability risks involved in executing arbitrary code on one of *your* systems. The existence of mobile code whose origin and execution characteristics are typically not well known necessitates the enforcement of strict security controls to prevent Trojan horses and other unanticipated effects. In certain cases, it may be desirable to provide repeated reauthentication and validation, and revocation or cache deletion as needed. (See Section 8.4.2.) When combined with digital signatures and proof-carrying code to ensure authenticity and provenance, dynamically linked mobile code provides a compelling organizing principle for highly survivable systems.

In principle, properly implemented environments for executing mobile code can contribute to survivability in various ways:

- Enable the execution of machine-independent trustworthy programs that have been carefully analyzed. Thus, the paradigm becomes not just write-once run-anywhere (WORA), but rather write-once, verify-once, and run-anywhere (WOVORA) . This can greatly enhance survivability by reducing the otherwise enormous task of verifying many different versions of the same code.
- Enable a distinction between the execution of programs on intrinsically unreliable and unsecure sites from execution on reliable and secure sites. In the sense of multilevel survivability, critical operations should not depend on code or execution on less trustworthy sites. For an MLX concept to be enforced, some measure of subsystem survivability must exist from which the aggregate survivability can be inferred or derived.

A highly survivable overall mobile-code architecture can be aided by a combination of trustworthy servers, encrypted network traffic, digital signatures, proof-carrying code, and other components and concepts discussed in Section 8.3. Three contemporary doctoral theses provide important contributions to the establishment of such an architecture:

- The formal-methods language-centric considerations of Drew Dean [82], exploring the type-safety of dynamic linking in Java-like environments
- The system-protection-centric work of Dan Wallach [347], considering the usefulness of access controls in what is called a security-passing-style architecture with well-defined semantics, which embodies some of the better attributes of capability-based systems and is specifically oriented toward mobile code, emerging as a generalization of his earlier work on stack inspection
- The proof-carrying code approach of George Necula [213], which for a given mobile-code segment provides an associated proof that can be dynamically and efficiently proof-checked to verify that it corresponds with the given code segment and that the code segment has a stronger sense of integrity than otherwise

Background on understanding code mobility rather independently of survivability and security issues is given in a useful article by Fuggetta et al. [100] (in a special issue of the *IEEE Transactions on Software Engineering* on mobility and network-aware computing). Formal methods are also particularly relevant to mobile code, because of the critical dependence on type safety – for example, the formalization of dynamic and static type checking for mobile code given in [289].

An extraordinary compilation of articles on various aspects of the mobile-code paradigm has been assembled by Giovanni Vigna, and published by Springer Verlag [343]. This book (which contains copious references) reflects most of the potential problems with mobile code, and suggests numerous approaches to reducing the risks. Considering the enormous potential impact, this book is mandatory reading for anyone trying to use the mobile-code paradigm in supposedly survivable systems. Following is a brief summary of the book.

- Part I considers theoretical foundations, and includes papers by Chess [66] on basic issues, Riordan and Schneier [290] on key generation, Volpano and Smith [345] on constraints that should be placed on programming languages for mobile code, and Sander and Tschudin [306] on the roles of cryptography in software-based approaches to secure mobile code.

- Part II considers relevant security mechanisms. Necula and Lee [215] further elaborate on the proof-carrying code concept, giving a prototypical design and characterizing its implementation. Hohl [121] explores the role of code obfuscation (which makes reverse engineering difficult) in protecting against malicious code. Berkovits et al. [37] consider roles of authentication: certification that a server has proper authority, allocation of least privilege, and state appraisal to ensure integrity. Vigna [342] considers the use of execution traces and cryptography to detect attacks against code changes, state changes, and execution flow of mobile code.

- Part III considers some specific mobile-code systems, Gray et al. [109] in D'Agents (formerly called Agent Tcl), Karjoth [142] in IBM's Aglets system, and Gong and Schemers [108] in protecting and cryptographically signing Java objects – signing, sealing, and guarding.

- Part IV considers the secure handling of mobile code on the World Wide Web. Ousterhout et al. [246] describe Safe-Tcl. DePaoli et al. [81] summarize the vulnerabilities in various so-called "secure" Web browsers.

8.4.1 Confined Execution Environments

The notion of a confined execution environment goes back at least as early as Multics. The nested Multics rings of protection were useful for protecting the system against its applications and protecting its applications against its users. However, the rings were also relevant to system survivability; a problem in ring 1 could not bring down the system (with critical code in ring 0) but might crash a user process, whereas a problem in ring 2 might abort a user command but would not crash the user process.

Important subsequent research came from Michael Schroeder [316, 318] (his doctoral thesis on domains of protection and mutual suspicion grew out of the Multics project) and Butler Lampson [156], with later work by Paul Karger [140] on preventing Trojan horses in a conventional access environment (that is, not multilevel secure). Ideally, it should be possible to control execution in such a way that nothing adverse can possibly happen.

The Java Virtual Machine (JVM) is an example of an execution environment designed to encapsulate the execution of code that can be dynamically obtained and loaded from arbitrary sources, subject to suitable security controls. Together with the Java Development Kit (JDK) [105, 107], JVM takes a significant step toward limiting bad effects that can take place in execution of an applet obtained from a potentially untrustworthy site. This is an example of a controlled execution domain whose intent is to radically limit what can and cannot be done, irrespective of the source of the applet. Systems designed to support secure and reliable execution of trustworthy mobile code can have an inherent potential stability. However, JVM is not yet a total solution, in that it is defined only in terms of single-user systems; it does not provide protection of one user from another simultaneous user.

The specific execution environments provided by Java, the Java Virtual Machine, and the Java bytecode have some serious potential security problems, largely attributable to the enormity of the code base and the fact that a very large portion of that code must be considered to be within the effective trusted computing base — which to a first approximation includes most of the run-time support, the bytecode verifier, the local operating system, the browser software, the servers from which code is obtained, and the networking software. Although this enormous security perimeter could be shrunk somewhat by techniques discussed below, the security perimeter for JVM applet security is very large.

In concept, many problems can be made worse by the presence of mobile code in heterogeneous networked systems. However, a well-engineered and properly encapsulated virtual machine environment has the potential of overcoming many of the risks that might otherwise arise in the use of arbitrary programming languages and the execution of arbitrary code. We believe that the mobile-code paradigm has enormous potential with respect to survivability (and the potential to withstand forced system crashes, loss of security, accidental outages, and so on), because of the roles it can play in inducing a survivable architectural structure. But that in itself forces us to think about the problems it raises.

There is of course a conflict between the desire to provide extensive functionality and the need to constrain or confine the functionality to make it secure — in order to help the overall computer-communication environment be survivable under attacks.

Of particular relevance here are the analyses of Drew Dean [82] and Dan Wallach [347] of the Java Security Manager (JSM), which is intended to be a security reference monitor that mediates all security-relevant accesses. A reference monitor is supposed to have three fundamental properties: (1) it is always invoked (nonbypassability), (2) it is tamperproof, and (3) it is small enough to be thoroughly analyzed (verifiability). Unfortunately,

1. The JSM is not always invoked. Programmers must remember to call it. If they do not, the default is that access is granted.

2. The JSM is not tamperproof. The type system can be compromised, which in turn can undermine the JSM security.

3. The JSM has no formal basis, and its complexity led to flaws in security policies embedded in JDK 1.0 and Netscape Navigator 2.0's SecurityManager.

In addition, both Dean and Wallach note that the Java language itself and its implementations do not have any auditing facility, and thus completely fail to satisfy the TCSEC accountability and auditing requirements.

8.4.2 Revocation and Object Currency

One of the problems associated with the mobile-code paradigm is that it is a *pull* mode rather than a *push* mode of operation. It could be advantageous to have subsequent improvements automatically downloaded, although that also creates potential integrity problems. Furthermore, there are cases in which it may be desirable or even necessary to revoke instantaneously all accesses to existing copies of a particular version of a program or data. However, existing browsers generally prefer locally cached versions to newer versions. The instantaneous revocation problem was investigated in the 1970s in the context of capability-based architectures (e.g., [92, 94, 104, 137, 141, 235]), beginning with David Redell's thesis [286, 287].

Under Redell's scheme, revocable access requires an extra level of indirection through a designated master capability, so that revocation of all copies of a given object could be effected simply by disabling the given master capability. (We could also contemplate a distributed set of equivalent capabilities that could in a practical sense be disabled simultaneously.) To achieve a similar effect in the context of the WOVORA mobile-code paradigm without undermining the performance benefits that result from caching, some sort of compromise push-pull mechanism is needed to ensure the currency of a locally cached object. Although instantaneous revocation seems intrinsically incompatible with local caching, various alternatives exist. One approach would be a single currency bit that is updated periodically, and checked whenever access from a cached version is attempted – forcing deletion of the cached object through dynamic reloading whenever the currency bit has been reset.

8.4.3 Proof-Carrying Code

The basic work on proof-carrying code (Section 5.8) comes from George Necula [213]. Each code module carries with it proofs about certain vital properties of the code. The validity of the proofs can be readily verified by a rather simple and relatively fast proof checker. If the proofs indeed involve critical properties, in principle any adverse alterations to the code (malicious or otherwise) are likely to result in the proofs failing.

8.4.4 Architectures Accommodating Untrustworthy Mobile Code

The survivable execution of untrustworthy mobile code depends on the successful isolation of the execution, preventing contamination, information leakage, and denials of service. What is needed in system and network architectures involves a combination of language-oriented virtual machines as in JVM [107], sandboxes [105, 179], differential dynamic access controls [348], mediators, trusted paths to the end user, less permissive bytecode verifiers, cryptography [307], and whatever authentication, digitally signed code, proof-carrying code [213, 214], and other infrastructural constraints may ensure that the risks of mobile code can be controlled. Not surprisingly, many of these requisite mechanisms are desirable for most meaningfully survivable environments, but the desirability of the mobile-code paradigm makes the potential vulnerabilities much more urgent.

8.5 Structural Architectures

A fundamental challenge in establishing architectures and configuring highly survivable systems and networks is to *constructively* use the structuring principles (Section 8.1) and architectural structures (Section 8.2) discussed above. Representative types of architectures are discussed next. Note that the mobile-code paradigm (Section 8.4) and the multilevel-survivability paradigm can be compatibly implementable within the same architecture – and indeed should be, considering the rampant popularity and enormous advantages of mobile code. However, the existence of mobile code forces us to confront problems that otherwise have lurked in the shadows for many years.

Because multilevel systems are less closely allied with what is commercially available today, and because our multilevel concept draws heavily on single-level components, the single-level concept is considered first.

8.5.1 Conventional Architectures

We consider next the simpler case of conventional single-level systems and networks, and attempt to define precisely which components of a structural architecture must be trustworthy with respect to each of the various dimensions of trustworthiness — for example, integrity, confidentiality, prevention of denial of service and other aspects of availability, guaranteed performance, and reliability. Table 8.2 summarizes some of the primary architectural needs that can contribute to overall survivability, in response to the identified limitations of Table 8.1. Throughout Table 8.2, it is evident that there is a pervasive need for good cryptography, by which is implied strong algorithms whose implementations and system embeddings are properly encapsulated, nonsubvertible, tamperproof, and reliable.

8.5.2 Multilevel Survivability with Minimized Trust

In considering the attainment of system-, network-, and enterprise-wide multilevel survivability (including appropriate MLS, MLI, MLA) without multilevel-secure end-user systems, we draw heavily on past work at SRI [241, 280] and ongoing work at NRL (e.g., [138]).

The basic strategy is conceptually simple. It mirrors some of the early work on multilevel-secure systems, with several fundamental differences:

- The overall system architecture makes distinctions among the various dimensions of trustworthiness noted in Section 8.5.1, with respect to integrity, confidentiality, prevention of denial of service, other aspects of availability, and guaranteed performance.
- The overall system architecture attempts to minimize the extent to which the various subsystems must be multilevel trustworthy (with respect to each of the specific

Table 8.2: Architectural Needs

Functionality	**Integrity**	**Confidentiality**	**Availability**	**Reliability**
User PC/NC OS, application code, browsers	Run-time checks, cryptographic integrity seals, accountability	Access controls; authentication, trusted paths, good encryption	Alternative sources, system fault tolerance	Constructive redundancy, reliable hardware
Networking and protocols	Better protocols, sound embeddings good encryption, tamperproofing	Better protocols, sound embeddings good encryption	More-robust defensive protocols, embeddings	More-robust defensive protocols, embeddings
Cryptography (for integrity, authentication, encryption)	Tamperproof and nonsubvertible implementations	Robust algorithms and protocols, nonsubvertible implementations	Dedicated hardware, sensible U.S. crypto policy!	Trustworthy sources, superimposed error correction
Authentication subsystems (e.g., with strong crypto)	Spoof proofing, replay prevention, crypo tokens, tamperproofing	One-time crypto-based tokens, in some cases biometrics	Alternative fault-tolerant authentication servers	Distributed consistency, redundancy in hardware
Trusted paths and resource integrity	Trusted path to users and servers, integrity as in user OSs (above)	Trusted path to users and servers, good encryption	Dedicated connections, alternative trusted paths	Self-checking, fault tolerance, dedicated circuits
Servers (file, ftp, http, e-mail, etc.)	Superior security, good encryption, authentication, better protocols, tamperproofing	Superior security, good encryption, authentication, better protocols	Mirrored file servers, robust fault-tolerant protocols	Constructive redundancy, self-checking protocols
Monitoring, analysis, response	Tamperproofing, nonbypassability, avoidance of overreactions	Enforcement of privacy concerns (much sensitive data involved)	Continuity of service, strong connectivity, self-diagnosis	Self-checking, fault tolerance, coordinated network management

dimensions), particularly by avoiding the need for trustworthiness in end-user systems over which there is no control or no expectation of multilevel trustworthiness, and by concentrating the enforcement of multilevel trustworthiness where it is absolutely essential. The basic architecture assumes that most of the end-user systems operate at a single MLS level at a time, and that most of the MLS enforcement is in the servers — with possibly some trustworthy (but not MLS trustworthy) local server interface software and possibly hardware co-processors, for example, as outboard cryptographic engines.

- The concept of monolithic multilevel kernels and TCBs as the basic building blocks is avoided, instead considering mostly multiple single-level systems and concentrating on a few selected trustworthy servers and other critical components.
- Generalized dependence is explicitly accommodated, on one hand avoiding the serious practical limitations of having to enforce strict multilevel partial orderings locally, and on the other hand allowing for trustworthy mechanisms to mediate among potentially less trustworthy components.
- All sensitive information and media entities (data, programs, network channels, and other definable resources) necessary for system survivability are potentially labeled with appropriate MLX labels (incorporating vectored MLS, MLI, MLA sublabels only where appropriate), giving all potentially critical subsystems explicit measures of trustworthiness. Any subsystem or component not explicitly labeled is assumed to be minimally trustworthy (that is, potentially, completely untrustworthy unless specific caveats are included).

Given this type of architectural structure, a relatively simple informal analysis can determine whether it is at all likely that the architecture can enforce the desired partial orderings dynamically. In other words, are there any gross violations of MLX dependence on less trustworthy subsystems? If so, can generalized dependence in some way adequately overcome the potential violations? Formal methods are not required in the basic stages of defining the architecture, although they could be useful later on.

Overall, we should not expect that, apart from MLS (which may be fundamental to certain applications), there would be a rigorous enforcement of strict partial ordering among the other attributes of MLX (namely, MLI and MLA) throughout the entire enterprise, and rather that mechanisms invoking generalized dependence can compensate for what would otherwise be violations of partial ordering.

8.5.3 End-User System Components

One of our most fundamental issues concerns the extent to which trustworthy systems can be developed despite the presence of end-user systems of varying degrees of untrustworthiness. This issue is very important in single-level systems (Section 8.5.1), and is even more important in the context of multilevel systems with minimized trustworthiness (Section 8.5.2). The following questions relate to end-user access to networked distributed environments that are intended to be highly survivable:

- What if the operating system security on an end-user system is not trustworthy (in some particular respects)? In that case, a masquerader or mistakenly trusted legitimate user might create considerable havoc on that system. For example, it is not wise to use a subvertible operating system for storing any cryptographic keys essential to local and network operations. Even if the keys are stored on a smart-card or similar device, flaws in the operating system are likely to make the cryptographic implementations subvertible or to make the unencrypted information accessible. Thus, the system architecture must explicitly assume the lack of trustworthiness, wherever that is a potential problem.

  In the case of multilevel operations, it might be deemed acceptable to assume that the user of a potentially untrustworthy end-user system could operate at a permissible secrecy level selected by its (apparent) user — if user authentication can be done in a nonsubvertible way that provides adequate trustworthiness, and if there is some sort of trusted path to the desired operating system (including integrity checks to ensure that the operating system had not been subjected to tampering). For example, a level of user authenticity could be provided by physical enclosures and personal recognition, or by an out-of-band biometric technique or a trusted-path cryptographic authentication with dedicated nontamperable hardware. If that assumption is not justifiable, the particular user of that end-user system would have to operate at an unclassified level. In any case, it must be realized that high-end authentication such as nonforgeable and nonreplayable biometrics may still be attacked by compromises from within and below. It must also be remembered that multilevel security does not address integrity issues (particularly those encompassed by multilevel integrity).

- What if the authentication of users on the end-user systems is not trustworthy? Then a masquerader can create considerable havoc from a local workstation, personal computer, or network computer. Additional authentication becomes essential before any further nonlocal access is permitted. Clearly, if users are allowed to have a single sign-on (that is, no further authentication as they move further along into networked systems), the end-user systems and other systems along the way must be substantially more trustworthy so that the authentication can itself be more trustworthy. However, single sign-on is a very bad idea if operating system security along the way and end-user authentication are not impeccable throughout. In cases in which the integrity o¡f the authentication is questionable, the system architecture must explicitly assume the lack of trustworthiness.

  In the case of multilevel operations, the particular user of that end-user system would have to operate at an unclassified level, or else submit to authentication by a trusted multilevel authentication server before any further nonlocal access is possible. If the local authentication is not trustworthy and the local operating system is not trustworthy, no access to multilevel resources should be permitted.

- What if the networking software on the end-user systems is not trustworthy? Again, the system and network architectures must explicitly assume the lack of trustworthiness.

In the case of multilevel operations, no further access should be permitted if any multilevel security is required across the network, or if the integrity of the networked remote entity cannot be assured, or if an entity-in-the-middle attack is possible.

- What if the encryption on the end-user systems can be bypassed or compromised? In that case, cryptography used for encrypting stored information can be compromised, encryption used in networking can be compromised, keys used for integrity checking may be spoofed, and authentication keys may be co-opted.

  In the case of multilevel operations, no further access should be permitted.

- What if the trusted path to the end-user systems can be compromised? In that case, fixed passwords can be trivially captured, although one-time passwords would be of no use (as long as they are locally generated and not concurrently reusable at other locations).

  In the case of multilevel operations, no further access should be permitted.

Thus, we are faced with essential trade-offs. If the local end-user operating systems and their trusted paths cannot be trusted, trustworthiness must not be assumed and the architecture must transfer trustworthiness to selected servers — where permitted. If the local authentication cannot be trusted, trustworthiness must be transferred to authentication servers. If the local networking software cannot be trusted, then trustworthiness must be transferred to selected network servers. On the other hand, if certain servers are not sufficiently trustworthy with respect to certain dimensions, then again trustworthiness in those dimensions must be transferred to servers that are more trustworthy.

If multilevel security is to be enforced, a sufficiently single-level secure local end-user system is necessary, nonbypassable local end-user authentication is necessary, multilevel-trustworthy networking is necessary even if local operation is single level (although cryptographic techniques can be used to ensure that if keys are distributed according to MLS requirements, no adverse flows can arise), and the trusted path and local system integrity must be noncompromisible.

Certain of the dimensions of survivability are more critical than others. For example, system integrity is generally paramount. If system integrity can be subverted, then it is usually easy to subvert confidentiality, availability, and reliability as well. On the other hand, denials of service can often result (whether intentionally perpetrated or accidentally triggered) without first subverting system integrity. Thus, it is advisable to consider each dimension in its own terms to determine the extent of the interdependencies.

By layering the mechanisms for protection, fault tolerance, and other aspects of survivability, and invoking the notion of generalized dependence, we might hope that a sufficiently survivable system could eventually be attained. However, access to sensitive MLS data should not be permitted whenever the end-user authentication cannot be guaranteed (with reasonable certainty), and also whenever the local end-user operating system can be compromised. Strict dependence on less trustworthy MLI resources should be avoided in any event.

Chapter 9

Implementing and Configuring for Survivability

You cannot make a silk purse out of a sow's ear.
Old saying

The architectural structures analyzed in Chapter 8 can be effectively implemented, and survivable systems can be effectively configured using some commercially available components plus the additional subsystems characterized in Chapter 5 to fill the gaps identified in Chapter 4. Whereas the proverbial silk purse is clearly unattainable from the sow's ear (despite a few system purveyors who would have you believe otherwise), it must be recognized from the outset that there will be remaining risks no matter what we do, because we are living in the real world rather than some idealized fantasy world. The challenge is to minimize those risks by relying on an architecture that is structurally sound, implementations that are robust where they need to be robust, operational practice that does not undermine the given requirements, and real-time analysis tools that can rapidly identify early threats to survivability and respond accordingly.

9.1 Developing Survivable Systems

Finally, following the foregoing paper discussion, we are ready to put the pieces together. A somewhat simplistic summary of the desired process is as follows:

1. The mission requirements must first be mapped onto a specific subset of the generic requirements, and some risk analyses done to ensure that those mission requirements are indeed adequate.

2. Given the specific requirements, a sketch of the anticipated architecture should be carried out.

3. The detailed architecture should then be fleshed out and documented sufficiently to enable a top-level examination of whether the functional and performance requirements are attainable.

4. Assuming that a review of item 3 is acceptable, a detailed plan for system evolution should be made, and the detailed system design should be carried out according to that plan.

5. A detailed implementation and test plan should be developed, and the system implemented and tested according to that plan. The implementation plan should address module composability with respect to integration, testing, and configuration management.

9.2 A Baseline Survivable Architecture

A suitable architecture for survivability might typically be one that encompasses those of the following desiderata deemed suitable for the given application in the case of dedicated systems, or the full range of expected applications in the case of systems that are more general-purpose.

- Take an open-system approach accommodating components of multiple vendors, different components from any particular vendors, different hardware platforms, multiple protocols, and different data formats.

- Facilitate evolutionary modifications of hardware, operating systems, application software, addition of new protocols and new implementations, and indeed total replacement of entire subsystems.

- Minimize the scope of trustworthiness, with a mixture of high-integrity components such as firewalls and necessarily trustworthy servers on one hand, and only partially trusted or completely untrusted entities on the other hand, with various intermediate degrees of trustworthiness as required.

- Include trusted paths wherever necessary, particularly whenever a local workstation must be trusted, when servers must be maintained, and generally whenever critical software must be upgraded.

- Provide systemic authentication wherever necessary, avoiding reusable fixed passwords and also avoiding single-sign-ons except within sufficiently trustworthy enclaves (if any such enclaves can actually exist, which seems very doubtful!).

- Provide robust encryption for integrity, authentication, confidentiality throughout, wherever appropriate, avoiding key-recovery and key-escrow schemes wherever possible.

- Facilitate the use of demonstrably trustworthy mobile code as appropriate, and otherwise constrain the presence of untrustworthy mobile code (perhaps in a properly encapsulated sandbox or otherwise confined environment that cannot contaminate its peers or hierarchical dependents).

- Include extensive mechanisms to hinder denial-of-service attacks.

- Accommodate generalized dependence whenever a strict dependence cannot be ensured.
- Accommodate the MLS and MLI whenever essential, preferably by resorting to a limited-trustworthiness MSL/MLS approach such as that of Proctor and Neumann [280] (see Section 8.2).
- Accommodate the spirit of multilevel survivability locally where it is appropriate, without attempting to enforce global MLX properties (while taking advantage of generalized dependence).
- Provide extensive coverage of vulnerabilities and threats that can be prevented, and further coverage of threats that when exploited can still be tolerated to some reasonable extent.
- Include comprehensive facilities for real-time monitoring, real-time analysis, and real-time response that aid in the maintenance of survivability and its subtended requirements at various layers of abstraction, from hardware to operating systems to servers to applications to entire networked enterprises. These facilities should be capable of detecting live threats that failed, that could not be prevented, and that could be tolerated at the moment but that might represent concerns in the future.

Detailed analysis of the candidate architecture is then needed to evaluate the appropriateness of the architecture, and detailed analysis of the feasibility of its successful implementation is needed to determine whether it is worth pursuing the particular architecture further. This is clearly an iterative process whenever the analysis determines inadequacies in the candidate architecture. In some cases, it may be appropriate to pursue alternative candidate architectures or variants thereof in parallel — at least until most of those alternatives can be discarded in favor of clear winners.

Considerably more detail will emerge during the second phase of this project, as specific architectural approaches are fleshed out and their implementation strategies are considered.

Chapter 10

Conclusions

Learning is not compulsory.
Neither is survival.
W. Edwards Deming

The currently existing popular commercially available computer-communication subsystems are fundamentally inadequate for the development and ready configuration of systems and networks with critical requirements for generalized survivability. Numerous good ideas exist in the research community, but are widely ignored in commercial practice. However, although it is theoretically possible to design dependable systems out of less-dependable subsystems or to design more-dependable critical components, it is in practice almost impossible to achieve any predictable trustworthiness in the presence of the full spectrum of threats considered here — including incorrect or incomplete requirements, flawed designs, flaky implementations, and noncooperating physical environments offering electromagnetic interference, earthquakes, massive power outages, and so on. Furthermore, the almost unavoidably critical roles of people throughout these systems and networks raise serious operational questions — especially relating to less-than-perfect individuals who may be dishonest, malicious, incompetent, improperly trained, disinterested, or who might in any way behave differently from how they would be expected to act in an assumed perfect world. These and many other considerations that are naturally subsumed under our notion of generalized survivability make the problems addressed here extremely challenging, important, and timely.

The challenge here is to do the best we can in the foreseeable future and to characterize steps that must be taken that will enable us to achieve better systems in the more distant future. There is still a lot to learn about survivability and how to attain it dependably. We hope that this report will be a significant step in that direction.

Unfortunately, the quest for simplicity and easy answers is pervasive, but very difficult to combat. In this report, we attempt to address the deeper issues realistically and to inspire much greater understanding of those issues.

10.1 Recommendations for the Future

Our main recommendations are summarized here. Specific directions for research and development are discussed in Section 10.2.

- We must establish detailed generic requirements for survivability and its subtended properties that can be directly invoked in system developments and procurements, or else mapped into applicable requirements based on specific mission requirements.
- We must develop new network and distributed system protocols and extend existing protocols (where appropriate) in anticipation of the creation of highly survivable, secure, and reliable information infrastructures.
- We must establish a library of demonstrably sound robust procedures that support the concept of generalized dependence, enabling not entirely trustworthy components to be used in applications where greater trustworthiness is required.
- We must find ways to encourage commercial system developers to bring more of the good research and development results into the commercial marketplace, and to develop some higher-assurance and higher-functionality components that are currently missing or inadequate.
- We must find ways to encourage system developers to increase the survivability, security, and reliability of their standard products.
- We must establish and consistently use sound cryptographic infrastructures for authentication, certificate authorities, and confidentiality, and avoid implementations that are intrinsically compromisible. This involves algorithms, software designs, and implementation.
- We must pursue more widespread use of nonproprietary open-source software and nonproprietary interfaces, and provide for mechanisms for trustworthy distribution of code, including robust mobile code — so that even though it may be freely acquired, it is nevertheless with high probability genuine and untampered.
- We must establish a collection of open-system architectural components that are compatible with higher-layer requirements for survivability and that have strong interoperability properties, enabling them to be readily composed into survivable systems and networks.
- We must pursue research and development on practical system composability that is also strongly based theoretically.
- We must refine and make practical the ongoing R&D efforts for monitoring, analyzing. Responding to system and network anomalies, we must generalize them from merely intrusion-detection systems, so that they address a broad range of survivability-related threats, including reliability problems, fault-tolerance coverage failures, and network classical management.

- We must find ways to introduce the concepts of this report pervasively into university curricula and other institutions' training programs, as suggested in the appendix.

10.2 Research and Development Directions

Section 5.15 considers the role of research and development. This section outlines some specific R&D directions for the future.

- Define, develop, and implement a set of robust networking protocols that systematically and comprehensively encompass the necessary requirements for survivability and its subtended attributes. (We believe that because of the increasing inadequacy of the existing protocols in light of massive bandwidth increases, today's protocol must be revisited anyway. Thus, we strongly urge taking advantage of that opportunity to ensure that survivability is not ignored in the process of protocol modernization.)

- Establish a definitive set of generic system requirements and elaborate on the process of mapping mission requirements into appropriate specific subsets of generic requirements. Establish measures for determining the adequacy of those requirements and the adequacy of subsequent implementations. Carry this process out explicitly for several specific applications such as the Tactical Internet.

- Design detailed open-system architectures for robust mobile code, and develop prototype implementations.

- Establish a comprehensive foundation for mobile code, specifying necessary infrastructural requirements on operating systems, browsers and servers, boundary protection, network protocols, programming languages, system composition, and other factors.

- Develop subsystems and components that can be readily integrated within the open-system architectures, such as boundary controllers, network servers, authentication servers, and mobile-code servers and constrained execution clients.

- Conduct research on specific aspects of generalized dependence, developing a general framework for accommodating it in system design and development. This research should address the mechanisms enumerated in Section 1.2.5. Although it should include the currently popular notion of wrappers that intend to mask flaws and overcome attacks, it should not ignore some of the other promising mechanisms for enhancing trustworthiness. It should also honestly address that fact that wrapper technology is susceptible to compromise from within and from below.

- Conduct research on predictable subsystem composition, by which the comprehensive realistic properties of the resulting composed system can be derived. Characterize properties of interfaces such that the compositions do not cause any adverse side effects. This work must encompass the full spectrum of survivability requirements, not just a very narrow band as has been the case in the past. In particular, it must encompass generalized dependence rather than just strict dependence on at-least-as-trustworthy components. It must also provide explicit conditions that must be satisfied by implementations and indicate how those conditions can be verified — statically or dynamically.

- Develop interoperable systems for monitoring, analyzing, and responding to system and network anomalies, encompassing security, reliability, and generalized survivability threats and classical network management.
- Explore the process of dynamic system configuration compatible with the selected subset of requirements. This should include both human-driven and autonomous reconfiguration. The process of altering system configurations without adversely compromising survivability should be at least partially automated, with a mixture of configuration-time and run-time checks and self-diagnostic techniques to ensure that the given requirements continue to be satisfied.
- Conduct research in dynamic adaptability of systems that does not compromise or seriously diminish overall system and network survivability. Although dynamic adaptability is similar in concept to dynamic reconfiguration, it has various aspects that need to be considered in their own light, particularly those involving the stability of autonomous actions in times of duress. However, research in dynamic adaptability should be coordinated closely with the issues involved in dynamic configuration.

10.3 Lessons Learned from Past Experience

> *[O]ur heads are full of general ideas that we are now trying to turn to some use, but that we hardly ever apply rightly. This is the result of acting in direct opposition to the natural development of the mind by obtaining general ideas first, and particular observations last; it is putting the cart before the horse. ... The mistaken views ... that spring from a false application of general ideas have afterwards to be corrected by long years of experience; and it is seldom that they are wholly corrected. That is why so few men of learning are possessed of common sense, such as is often to be met within people who have had no instruction at all.*
>
> Arthur Schopenhauer,[1] 1851

Many lessons can been gleaned from experience with past system developments, both successful and unsuccessful. These experiences can help us to calibrate the appropriateness of the various principles scattered throughout this document.

The work of Henry Petroski [268, 269] (a civil engineer at Duke University) is noteworthy. Petroski has often observed that we tend to learn very little from our successes and that we generally can learn much more from our failures. Unfortunately, the experiences documented extensively by Neumann [228] suggest that the same mistakes tend to be made over and over again — particularly in computer-related systems.

---

[1] Excerpted from *Parerga and Paralipomena*, 1851, as included in *Schopenhauer Selections*, edited by DeWitt H. Parker, Scribners, New York, 1928, with minor modernization of the translation by PGN.

Here are a few conclusions, in part tempered by watching the negative experiences in the on-line Risks Forum, and in part from highlighting the constructive aspects of some past system efforts. If Schopenhauer and Petroski are as fundamentally correct as it appears they are, we must learn more from our experiences, good and bad.

- Requirements must be sufficiently correct, consistent, complete, and precisely specified, from the very beginning of a procurement or development. Requirements must encompass not just the critical functional properties (such as survivability, security, and reliability), but also operational requirements (such as interoperability, reusability, and performance).
- The architectural fundamentals must be sound from the very beginning, and demonstrably capable of satisfying the given requirements. The design must be thoroughly specified. Survivability and its subtended requirements (notably security and fault tolerance) must be part of the overall conception rather than retrofitted after the fact.
- The implementation strategy must be realistic, and followed consistently throughout unless major flaws are identified and corrected.
- The implementation should be demonstrably capable of satisfying its specification.
- The system architecture and the entire development cycle must anticipate the needs of evolutionary growth, and must take significant advantage of what has been learned in research as well as what can be learned from experience in developing operating systems, networks, architecture, protocols, cryptographic implementations, software engineering tools, and other disciplines.

The following itemization of lessons learned amplifies some overall strategies for achieving highly survivable systems and networks, but is also applicable to less critical environments.

- Identify your long-range goals up front before trying to haggle over your short-range goals. It is often very difficult to retrofit measures that can overcome what you ignored initially, despite the notion of generalized dependence. This is especially true of security whenever the underlying infrastructure is not secure. It is also true of reliability and fault tolerance. It is only somewhat less true of a decent human interface that hides all the warts of a system, because in many cases you cannot hide the warts – they tend to shine through no matter how hard you try. On the other hand, if the system is really well designed (with abstraction, encapsulation, information hiding, strong typing, and other principled properties), then you may be able to provide a human interface that masks extraneous implementation detail. Nevertheless, complex systems in the hands of incompetent or inadequately trained personnel (including users, operators, system administrators, and security officers) inherently represent further risks. (The Aegis system involved in the Vincennes shooting down the Iranian Airbus is a good example of a bad user interface that could not overcome – and in fact exacerbated – some of the limitations of the system itself.)

- Fred Brooks' "Build one to throw one away" is not a good practice in general (and Fred has more recently noted that) – although the concept has some merits if one person has total control over both the throw-away prototype and the real-thing system, and if time and money are not at stake. Much better is to have a well-thought-out evolutionary strategy from the beginning, and be able to grow your prototype into what you really wanted in the first place. With modularity in distributed systems, well-defined interfaces, operability standards, and strong-willed management, this is a very powerful approach. Such an evolutionary approach can even overcome the lack of foresight in the original concept. However, the success of such an evolutionary strategy depends on a thorough *a priori* understanding of the survivability requirements, and also depends on the comprehensive adherence to those requirements throughout the development.

- Fred Brooks' first edition of *The Mythical Man Month* maintained that every programmer needs to understand the entire system. In the second edition [60], Brooks recants, stating that he was wrong and Parnas was right. Parnas's concepts ([72, 251, 250, 252, 260, 261, 253, 258, 254, 257, 255, 259], listed chronologically rather than numerically) of decomposition, modularization, abstraction, transparency, and information hiding indeed are very powerful, especially in structuring and designing large projects.

- Good software engineering practice may seem to be very difficult to achieve, but it is invaluable in the long run. It is usually worth the effort if your goals are nontrivial. For example, forget about merely trying to pretend that you are "object oriented" by using a few object-oriented tools. Understand the deeper benefits of abstraction, encapsulation, polymorphism, and inheritance in terms of the system structure they induce – and above all honor the concept of good design structure. (See the next item.)

- The quote from Jim Horning and Brian Randell at the beginning of Chapter 8 is vital. Furthermore, it is not enough merely to have good design structure. You must also ensure that significant structural boundaries (as in isolation kernels, domain separation, and calling semantics) do not get compromised by the implementation (if indeed you had good structure in the first place) – as in the case of domain violations, buffer overflows, unchecked parameters, and a raft of other bugs that result in blowing the integrity of the design. Architectural structures that inherently facilitate constructive modularity, predictable composition, evolvability, and interoperability should be cherished, because they provide enormous long-term benefits.

- Short-term or local optimization is a popular lure that is almost always counterproductive in the long run. The alternative does not have to be trying to achieve global optima; that is vastly too difficult. It is advantageous merely to be aware of and consider in balance the long-term issues before leaping to short-term conclusions. The long-term perspective often indicates that you should do things radically differently from what the short-term perspective suggests.

The relative roles of experiential knowledge and general principles are considered in the context of education in the appendix.

10.4 Architectural Directions

Chapters 8 and 9 provide guidelines, principles, and architectural structures for designing and implementing systems and networks with stringent survivability requirements. Experience tends to support the belief that highly principled and architecturally motivated designs have a much greater likelihood of converging on systems and networks that can successfully meet stringent requirements, and that can evolve gracefully to accommodate changing requirements. In particular, highly structured designs, domain separation, encapsulation, information hiding, cleanly defined interfaces, formal models of composition and interconnectivity, and many other concepts explored in this report can all have very significant payoffs.

10.5 Residual Vulnerabilities and Risks

Despite the best intentions on the part of the architects of systems and networks having strong survivability requirements, many vulnerabilities are still likely to remain. Hardware is always a potential source of system failures (and, potentially, physical attacks), either transient or unsurmountable without physical repair. The software implementation process is fundamentally full of risks. Operationally, knowledgeable system and network administrators are chronically in short supply, and their role in maintaining a stable environment is absolutely critical. In addition, opportunities are rampant for malicious Trojan horses and just plain flaky software — especially in mobile code and untrusted servers. Penetrations from outside will always be possible in some form or another, and misuse by insiders remains an enormous source of risks in many types of systems. Furthermore, because systems and networks tend not to be people tolerant, users are inevitably a source of risks — no matter how defensively the user interfaces may appear to be. These expected residual shortcomings must be anticipated. Real-time analysis for misuse and anomaly detection still remains desirable as a last resort, even in the best of systems.

10.6 Applicability to the PCCIP Recommendations

The recommendations of Section 10.2 provide a considerable sharpening of some generic recommendations of the report of the of the President's Commission on Critical Infrastructure Protection [182]. It is interesting to contrast what comes out of the survivability-driven concepts of this report with the funding areas suggested by the Commission:

1. Information assurance
2. Intrusion monitoring and detection
3. Vulnerability assessment and systems analysis
4. Risk-management decision support
5. Protection and mitigation

6. Incident response and recovery

The recommendations of our report address many research and development aspects of items 1, 2, 3, and 5, while at the same time focusing more broadly on survivability rather than just security. Item 6 is also important, although somewhat peripheral to the focus of our efforts. Because it does not create an architectural forcing function (other than the motherhood idea that we should reduce the number of vulnerabilities that must be reported), we simply assume that emergency response teams will exist in a more effective form than at present.

Item 4 causes us some concern. Although risk *avoidance* is in essence what this report is all about, and static risk assessment is important (see Section 5.12), the notion of decision support tools is potentially dangerous. Many of those tools that purport to *manage* risks actually encourage us to ignore certain risks rather than prevent them. However, those tools are often based on incorrect assumptions, and frequently ignore the interactions or dependencies among different requirements, components, and threat factors. Reliance on support tools to perform risk management is very dangerous in the absence of deep understanding of the risks, their causes, and their consequences. On the other hand, if we had that deep understanding, we might have better systems, and the risks could actually be avoided to a much greater extent — rather than having to be managed! Nevertheless, an interesting potential attempt to quantify risks and to balance defending against perceived threats with what can be considered acceptable risks is given by Salter et al. [304]. The Software Engineering Institute at Carnegie-Mellon University is working on a taxonomy of risks (`http://www.sei.cmu.edu`).

A general discussion of the risks of risk analysis itself can be found in a brief but incisive contribution of Bob Charette ([228], Section 7.10, pages 255-257). For example, overestimating the risks can cause unnecessary effort; underestimating the risks can cause disasters and result in reactive panic on the part of management and affected people; estimates of parameters are always suspect; perhaps most important, there is seldom any real quality control on the risk assessment itself.

10.7 Future Work

The exact focus of the second phase of this project, beginning immediately, will be subject to specific agreement in discussions with the Army Research Lab. It is likely to include directions such as the following:

- Expand further on the architectural directions as well as the research and development recommendations outlined in this report, leading to a level of detail sufficient to enable specific guidance to system developers with respect to the missing or deficient components, and to simplify considerably the challenges of implementing and configuring significantly more survivable systems and networks.
- Apply the refined architectural directions to the Tactical Internet environment, and evaluate the feasibility of the results.

- Investigate some of the most promising alternatives for the robustification of open-source software, and establish specific strategies for implementing them.
- Elaborate on the educational directions of the appendix that would enable the subject of this report to be taught within instructional programs within the Army, DoD, academic, and industrial mainstreams. As appropriate within the framework of the project (or in some cases outside of the scope of the project), we will contribute to teaching courses on survivability and participate in relevant DoD information survivability workshops.

10.8 Final Comments

We have outlined many concepts that are highly relevant to the specification, design, development, and operation of highly survivable systems and networks. The architectural directions pursued here integrate those concepts — particularly generalized dependence and generalized composition — and provide a strong basis for systems that accommodate mobile code, portable user platforms and robust execution platforms, minimal critical dependence on untrustworthy components, and highly reconfigurable and adaptive environments. Much work remains to be done to demonstrate the applicability of this approach, but we believe that we have broken some new ground.

This is not a case of Just Add Money and Stir. We still have a very long way to go. Serious understanding is needed of the depths of the problems and the urgent needs for solutions. Although some significant progress can be made in the short term, commitment to long-term advances is essential.

The words of Albert Einstein again seem pithy, this time in circumscribing the rather modest intent of the author of this report — despite the enormity of the underlying challenge:

> *Finally, I want to emphasize once more that what has been said here in a somewhat categorical form does not claim to mean more than the personal opinion of a man, which is founded upon* nothing but *his own personal experience, which he has gathered as a student and as a teacher.*
> Albert Einstein,[2] 1950

---

[2] *Out of My Later Years,* The Philosophical Library, Inc., New York, NY, 1950, p. 37.

Acknowledgments

We are indebted to CSL colleagues Jonathan K. Millen and Phillip A. Porras for many stimulating discussions. Jon also made some notable research contributions on the project [197, 198]. We owe thanks to Richard Drews Dean for many discussions related to this report. Drew spent two summers in the SRI Computer Science Laboratory and also maintained close contact with us during the time he was working at Princeton on his Computer Science PhD thesis [82].

We are very grateful to Paul Walczak for his valuable assistance and strong encouragement, and to Tony Barnes for his deep interest in the project and his long-term interactions [25]. Tony almost single-handedly stimulated awareness of survivability issues within the Army and the DoD in the period leading up to the initiation of the current project.

Appendix: Curriculum for Survivability

It is essential that the essence of this report be moved into the educational and institutional mainstream so that the valuable experiences of the past can be merged effectively with recent advances in computer and network technology, and thus encourage the development of truly survivable systems and networks in the future, and stimulate greatly increased awareness of the issues.

A.1 Survivability-Relevant Courses

The following plan for courses that might contribute to an educational program are specifically oriented toward a systems perspective of survivability. Two types of course programs are identified, one on a small scale, the second on a much broader scale. However, the two types are compatible, and the differences are not intrinsic. Relevant curriculum topics are identified in Table 10.1 for each type.

- Type 1: Creation of one new course. The single course would focus primarily on survivability-related issues, with specified prerequisites depending on the intended level of the course. Suggested single-course material is identified in the table with an "S" in the first column. Suggested appropriate prerequisite material would be selected from the items denoted by the sequential letters "A", "B", "C", "D", "E", based on the availability of existing course offerings.
- Type 2: Creation of a fully integrated curriculum. A relatively self-contained course program would integrate survivability-specific material with other parts of a normal computer-science and computer-engineering curriculum, achieving a coordinated program. An illustrative self-contained curriculum sequence could be configured out of material selected from items in the table from "A" through "S", plus the options of possible subsequent course material denoted by "T", "U", "V", "W" and possible practical projects subsumed under the designation "X" — depending on the academic level.

The level of detail and the nature of the material selected should of course be carefully adapted to the academic level and experience of the students (e.g., college undergraduates, university graduate students, industrial employees, reentry individuals being retrained for new careers), and the background, training, and experience of the teaching staff. Many

Phase	Topic	Sources
A	Fundamentals of Programming Languages	Prerequisite (Type 1) or integrated unit (Type 2)
B	Fundamentals of Software Engineering	Prerequisite (Type 1) or integrated unit (Type 2) (see X for possible follow-on)
C	Fundamentals of System Engineering	Integrated material (Type 1), but not widely taught today
D	Fundamentals of Operating Systems	Prerequisite (Type 1) or integrated unit (Type 2)
E	Fundamentals of Networking	Prerequisite (Type 1) or integrated unit (Type 2)
S	Introduction to survivability: concepts, threats, risks, egregious examples	Chapter 1(Type 1 or 2)
S	Specific Threats	Chapter 2 (Type 1 or 2)
S	Survivability Requirements	Chapter 3 (Type 1 or 2)
S	Systemic Deficiencies	Chapter 4 (Type 1 or 2)
S	Systemic Approaches	Chapter 5 (Type 1 or 2)
S	Evaluation Criteria	Chapter 6 (Type 1 or 2)
S	System Architectures	Chapters 8,9 (Type 1 or 2)
T	Advanced topics in systems, networks, databases, architecture, system and software engineering, etc.	Follow-on (Type 1 or 2); Pursue bibliography
U	Advanced topics in survivability, security, encryption, reliability, fault tolerance, error-correcting codes, etc.	Follow-on (Type 1 or 2); Pursue bibliography
V	Applications of formal methods to critical aspects of survivability	Follow-on (Type 1 or 2); Pursue bibliography
W	Management of development, quality control, risk assessment, human factors, robust open-source, etc.	Follow-on (Type 1 or 2), Pursue bibliography
X	Prototype development projects, ideally with collaborating teams, especially robustification of open-source software	Follow-on (Type 1 or 2), integrated with B sequence

Table 10.1: Survivability Curricula

advantages can result from integrating requirements for survivability and its subtended attributes such as reliability, security, and performance, early in a student's life. However, many of the subtleties of the system development process (e.g.,, team communication failures and the pervasiveness of system vulnerabilities) and many of the idiosyncrasies of procurement, configuration, and operation do not become meaningful to students until they have gained sufficient experience.

We make a distinction in the table between familiarity with programming languages and operating systems on one hand, and a deeper understanding of the principles thereof on the other hand. It is not adequate that students have merely been exposed to many different systems and languages. It is vital that they understand the fundamentals of those systems and what is really necessary in the future. Thus, the prerequisites or integrated units shown in the table for "A" through "E" are in the long run not necessarily the standard courses that exist today in their respective areas, but rather courses or units of courses that stress a grasp of the appropriate fundamentals. Nevertheless, exposure to some modern programming languages is highly desirable.

Ideally, an academic program incorporating survivability should have elements of survivability and its subtended requirements distributed throughout a considerable portion of the basic curriculum. In such an ideal world, those requirements would be addressed in the existing courses designated by "A" through "E" and "T" through "X" in Table 10.1.

In contrast, survivability is almost never addressed today, and security and reliability are typically specialty subjects, and then only in a few universities. Similarly, software engineering may be taught as a collection of tools, rather than as a coherent set of principles. As a result, from a practical viewpoint, it would be very difficult to achieve a fully integrated approach as an incremental modification to existing course structures. On the other hand, it would be relatively easy to initially introduce a single new course addressing the items denoted by "S" in the table, and then evolve toward the desired goal of a coherent integrated curriculum.

Thus, our basic recommendation is to start small with the single course focusing on the "S" items most specifically related to survivability, and then over time to encourage faculty members to allow the concepts of survivability to osmose, pervasively working themselves into the broader academic program — with particular emphasis on the design of operating systems and networking, and the use of programming languages and software engineering techniques to achieve greater survivability. In the process, the material in each course offering may change somewhat, including the material earmarked for the single type-1 course on survivability — some of which may tend to be distributed among some of the prerequisite courses.

One of the most fruitful areas for student projects involves the robustification of open-source software. The variety of approaches is enormous, the challenges are unlimited, and the opportunities for successful penetration into the real world are very exciting. The best results will find instant use on the Web. Collaborations with others can lead to continual improvements.

At present, no obvious textbooks can contribute directly to the intended breadth of the outlined survivability curriculum (other than perhaps this report, which in the first phase of the project has been primarily concerned with the fundamentals rather than the details — which are yet to follow in the second phase). However, there are various books that can be extremely useful in filling in some of the gaps — for example, addressing security (e.g., [271]), Java security and secure mobile code (e.g., [187] or, when available, [188]), and software engineering (e.g., [272]). Surprisingly, there does not seem to be an appropriately scoped modern book on fault tolerance, although there are some significant journal articles, such as [75] on distributed fault tolerance, and an early book on principles [17] that although out of print is still useful. Far-sighted good principles tend to remain good principles forever, despite changes in technology. However, the understanding and appreciation of those principles is highly dependent on their being illustrated by concrete examples. Furthermore, technological changes often tend to make optimization advice obsolete.

Many important articles should be mandatory reading for any students seeking a deeper understanding, quite a few of which are cited explicitly in this report. (See the References section introducing the Bibliography.) However, it is clear that someone needs to write an up-to-date textbook that could be used for the core portion of the proposed survivability curriculum. Perhaps this report will serve as the basis for such a book. On the other hand, books are often obsolete before they are published — which is why this report has focused primarily on principles and their underlying motivation, as well as why it is important to study the literature.

Unfortunately, the prevailing mentality among many younger researchers and developers is fairly troglodytic when it comes to earlier works: "If it isn't on the Web today, it never existed in the past." Many extremely important works in the literature tend to be forgotten. (One effort to resuscitate some historically significant efforts is the History of Computer Security Project, which has created CD-ROMs of seminal papers. See `http://seclab.cs.ucdavis.edu/projects/history` for further information.)

A.2 Applicability of Remote Learning

Some universities and other institutions are offering or contemplating courses taught on-line via the Internet, including a few with degree programs. There are many potential benefits, as the multimedia technology improves with respect to audio, video, and sophisticated graphics: teachers can reuse collaboratively prepared course materials; students can schedule their remote studies at their own convenience, and employees can participate in selected subunits for refreshers; society can benefit from an overall increase in literacy – and perhaps even computer literacy. On-line education inherits many of the advantages and disadvantages of textbooks and conventional teaching, but also introduces some challenges of its own:

- People involved in course preparation quickly discover that creating high-quality teaching materials is labor intensive, and very challenging. To be successful, on-line instruction requires even more organization and forethought in creating courses than normally, because there may be only limited interactions with students, and anticipating all possible options is difficult. Thoughtful planning and carefully debugged instructions are essential to make the experience more fulfilling for the students. Furthermore, for many kinds of courses, on-line materials must be updated regularly to remain timely.

- Major concerns arise regarding who owns the materials (some universities claim proprietary rights to all multimedia courseware), with high likelihood that materials will be purloined or emasculated. Some altruism is desirable in exactly the same sense that open-source software (see Section 5.9) has become such an important driving force. Besides, peer review and ongoing collaborations among instructors could lead to continued improvement of public-domain course materials.

- Administrators are likely to seek cost-cutting measures, such as firing qualified instructors in the common quest for easy answers, oversimplifying the content, and cutting costs through other misguided measures.
- Loss of interactions among students and instructors is a serious potential risk, especially if the instructor does not realize that the students are simply not getting it.

The last of these challenges can be partially countered by including some live lectures or videoteleconferenced lectures, and requiring instructors and teaching assistants to be accessible on a regular basis, at least asynchronously via e-mail. Multicast course communications and judicious use of Web sites may be appropriate for dealing with an entire class. However, the reliability and security weaknesses in the information infrastructures suggest that students will find lots of excuses such as the "Internet ate my e-mail" variant on the old "My dog ate my homework" routine. Inter-student contacts can be aided by chat rooms, with instructors trying to keep the discussions on target. Also, students can be required to work in pairs or teams on projects whose success is more or less self-evident.

E-education may be better for older or more disciplined students, and for students who do not expect to be entertained. It is useful for stressing fundamentals as well as helping students gain real skills. But only certain types of courses are suitable for on-line offerings – unfortunately, particularly those courses that emphasize memorization and regurgitation, or that can be easily graded mechanically by evaluation software. Such courses are also highly susceptible to cheating, which can be expected to occur rampantly whenever grades are the primary goal, used as a primary determinant for jobs and promotions. Cheating tends to penalize only the honest students. It also seriously complicates the challenge of meaningful professional certification based primarily on academic records.

Society may find that distance learning loses many of the deeper advantages of traditional universities – where smaller classrooms are generally more effective, and where considerable learning typically takes place outside of classrooms. But e-education may also force radical transformations on conventional classrooms. If we are to make the most out of the challenges, the advice of Brynjolfsson and Hitt [61] suggests that new approaches to education will be required, with a "painful and time consuming period of reengineering, restructuring and organization redesign..."

There is still a lack of experience with, and lack of critical evaluation of, the benefits and risks of such techniques. For example, does electronic education scale well to large numbers of students in other than rote-learning settings? Can a strong support staff including in-person teaching assistants compensate for many of the potential risks? On the whole, there are some significant potential benefits, for certain types of courses. We hope that some of the universities and other institutions already pursuing remote electronic education will evaluate their progress on the basis of actual student experiences (rather than just the perceived benefits to the instructors), and share the results openly. Until then, we vastly prefer in-person teaching coupled with students who are self-motivated — although there have clearly been some strongly positive experiences with videoteleconferencing.

If electronic materials are to be used in a survivability syllabus, we recommend starting modestly and then extending the offerings in a careful evolutionary manner.

A.3 Summary of Education and Training Needs

The combination of architectural solutions, configuration controls, evaluation tools, and certification of static systems is by itself still inadequate. Ultimately, the demands for meaningfully survivable systems and networks require that considerable emphasis be placed on education and training of people at many different levels – including high-level definers of high-level requirements, those who refine those requirements into detailed specifications, system designers, software implementers, hardware developers, system administrators, and especially users. The concept of keeping systems simple cannot be successful whenever the requirements are inherently complex (as they usually are). (Once again we recall the quotes from Albert Einstein given in Chapter 1.) Training large numbers of people to be able to cope with enormous complexity is also not likely to be successful. Although the mobile-code paradigm offers some hopes that education and training can be simplified, many vulnerabilities in the underlying infrastructure require human involvement, especially intervention in emergency situations. In short, our dictum throughout this report that "there are no easy answers" also applies to the challenges of education and training. There is no satisfactory substitute for people who are intelligent and experientially trained. But there is also no satisfactory substitute for people-tolerant systems that can be survivable despite human foibles. The design of systems and networks with stringent survivability requirements must always anticipate the entire spectrum of improper human behavior and other threats. We need intolerance-tolerant systems that can still survive when primary techniques for fault tolerance and compromise resistance fail, irrespective of unexpected human and system behavior. But above all we need people with both depth of experience and depth of understanding who can ensure that the established principles are adhered to throughout system development and maintained throughout system operation, maintenance, and use.

A.4 The Purpose of Education

We conclude this appendix with yet another succinct quote from Albert Einstein, who serendipitously summarized the primary aim of our intended efforts to bring survivability concepts into mainstream curricula:

> *The development of general ability for independent thinking and judgment should always be placed foremost, not the acquisition of special knowledge. If a person masters the fundamentals of his subject and has learned to think and work independently, he will surely find his way and besides will better be able to adapt himself to progress and changes than the person whose training principally consists in the acquiring of detailed knowledge.*
>
> Albert Einstein,[3] 1950

The wisdom of Einstein (embodied in quotations throughout this report) and Schopenhauer (in Section 10.3) emphasizes what should be the deeper purpose of any education relating to such a comprehensive subject as survivable systems and networks. Concerning Schopenhauer's view that experience must motivate the application of general principles, we must also keep in mind that folks who start out with only experience and no principles also tend to go astray. Our own holistic view on the subject can be summarized as follows:

- Teaching is not enough; there must be experiential learning.
- Experience is not enough; there must be guiding principles.
- Principles are not enough; there must be serious understanding.
- Understanding is not enough; there must a real commitment to robustness and dependability, and what it takes to achieve them.
- Teaching, experience, principles, and understanding may not be enough, but they are a good start. They must be embedded deeply into the system development process, and must consequently permeate system configuration, operation, maintenance, and use.

Perhaps the successive refinements of this report in combination with other materials noted on the following page — for example, [228] – can provide a useful starting place.

---

[3] *Out of My Later Years,* The Philosophical Library, Inc., New York, NY, 1950, p. 36.

Noteworthy References

Because this report addresses some of the most fundamental limitations of commercially available systems and what must be done to overcome those limitations, research is of critical importance to survivability — both the incorporation of ongoing research and the conduct of new research that can help to fill in the gaps. As a consequence, considerable emphasis is placed on research references in the following bibliography. Although new additions to the literature are continually emerging, we have attempted to focus on the primary references, and particularly those that might illuminate a highly principled approach to survivability.

We cite here a few references that have been particularly influential in this project, and that we consider to be of historical significance in understanding the importance of architectural structure and its implications. Although the following subset of references is largely concerned with security, it also provides many valuable insights with respect to achieving survivability:

- Multics' virtual memory, segmentation, paging, hierarchical access controls, symbolic naming and aliasing for files and input-output streams, dynamic linking, domain isolation, and the use of a higher-level system programming language in the 1960s [77, 78, 245], followed by a redesign and retrofit of the kernel to accommodate multilevel security in the 1970s [317]; two papers on lessons learned [70, 71]
- Schroeder's domains of protection [316], Lampson's notion of confined execution [156], and Saltzer and Schroeder's fundamental paper on protection [305]
- Dijkstra's THE system, with a hierarchical locking strategy to prevent interlayer deadly embraces [88]
- SRI's Provably Secure Operating System effort, an early object-oriented hardware-software operating system design, tagged capabilities, and non-kernel MLS, hierarchical abstraction, each layer formally specified [94, 235], an effort that also led to the Hierarchical Development Methodology (HDM) [293], the SPECIAL specification language [235], and Extended HDM (EHDM) [325]
- Rushby's isolation kernels [297]
- SeaView [83, 176, 178], for achieving a high-assurance MLS DBMS without requiring any trustworthiness for MLS in the DBMS via balanced assurance
- The Proctor-Neumann covert-channel-free MLS/MLI architectures, with no end-user MLS, of considerable potential interest today given the continued absence of COTS MLS systems [241, 280]
- A few early writings link security and reliability, such as Lampson in 1974 [157] and Dobson and Randell in 1986 [90, 282], and Neumann beginning at least in 1969 [221, 222, 223, 237, 224] (listed chronologically).

Bibliography

[1] M. Abadi. On SDSI's linked local name spaces. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, pages 98–108, Rockport, Massachusetts, June 1997.

[2] M. Abadi and A.D. Gordon. A calculus for cryptographic protocols: The Spi calculus. Technical report, Digital Equipment Corporation, SRC Research Report 149, Palo Alto, California, January 1998.

[3] M. Abadi and L. Lamport. Composing specifications. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, pages 1–41, REX Workshop, Mook, The Netherlands, May-June 1989. Springer-Verlag, Berlin, Lecture Notes in Computer Science Vol. 230.

[4] M. Abadi and K.R.M. Leino. A logic of object-oriented programs. Technical report, Compaq Systems Research Center, SRC Research Report 161, Palo Alto, California, September 1998.

[5] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, January 1996.

[6] H. Abelson, R. Anderson, S.M. Bellovin, J. Benaloh, M. Blaze, W. Diffie, J. Gilmore, P.G. Neumann, R.L. Rivest, J.I. Schiller, and B. Schneier. The risks of key recovery, key escrow, and trusted third-party encryption. *World Wide Web Journal (Web Security: A Matter of Trust)*, 2(3):241–257, Summer 1997. This report was first distributed via the Internet on May 27, 1997.

[7] H. Abelson, R. Anderson, S.M. Bellovin, J. Benaloh, M. Blaze, W. Diffie, J. Gilmore, P.G. Neumann, R.L. Rivest, J.I. Schiller, and B. Schneier. The risks of key recovery, key escrow, and trusted third-party encryption. `http//www.cdt.org/crypto/risks98/`, June 1998. This is a reissue of the May 27, 1997 report, with a new preface evaluating what happened in the intervening year.

[8] B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[9] J. Alves-Foss. The use of belief logics in the presence of causal consistency attacks. In *Proceedings of the Nineteenth National Computer Security Conference*, pages 406–417, Baltimore, Maryland, 6–10 October 1997. NIST/NCSC.

[10] E. Amoroso. *Intrusion Detection: An Introduction to Internet Surveillance, Correlation, Trace Back, Traps, and Response.* Intrusion.Net Books, 1999.

[11] D. Anderson, T. Frivold, and A. Valdes. Next-Generation Intrusion-Detection Expert System (NIDES). Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, SRI-CSL-95-07, May 1995.

[12] D. Anderson, T. Lunt, H. Javitz, A. Tamaru, and A. Valdes. Safeguard final report: Detecting unusual program behavior using the NIDES statistical component. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, 2 December 1993.

[13] M. Anderson, C. North, J. Griffin, J. Yesberg, and K. Yiu. Starlight: Interactive link. In *Proceedings of the Twelfth Annual Computer Security Applications Conference*, pages 55–63, San Diego, California, December 1996. IEEE Computer Society.

[14] R. Anderson and M. Kuhn. Low cost attacks on tamper resistant devices. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science, Security Protocols, Proceedings of 5th International Workshop*, pages 125–136, Paris, France, April 1997.

[15] R.H. Anderson. A "minimum essential infrastructure" (MEII) for U.S. defense systems: Meaningful? feasible? useful? In *Position Papers for the 1998 Information Survivability Workshop — ISW '98*, pages 11–14, Orlando, Florida, 28–30 October 1998. IEEE.

[16] T. Anderson and J.C. Knight. A framework for software fault tolerance in real-time systems. *IEEE Transactions on Software Engineering*, SE-9(3):355–364, May 1983.

[17] T. Anderson and P.A. Lee. *Fault-Tolerance: Principles and Practice.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1981.

[18] W.A. Arbaugh, J.R. Davin, D.J. Farber, and J.M. Smith. Security for private intranets. *IEEE Computer*, 31(9):48–55, 1998. Special issue on broadband networking security.

[19] W.A. Arbaugh, D.J. Farber, and J.M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 Symposium on Security and Privacy*, pages 65–71, Oakland, California, May 1997. IEEE Computer Society.

[20] W.A. Arbaugh, A.D. Keromytis, D.J. Farber, and J.M. Smith. Automated recovery in a secure bootstrap process. In *Proceedings of the 1998 Network and Distributed System Security Symposium*, San Diego, California, March 1998. Internet Society.

[21] A. Arora and S.S. Kulkarni. Component based design of multitolerance. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.

[22] A. Arora and S.S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *Proceedings of ICDCS*, 1998.

[23] J. Bacon. Report on the Eighth ACM SIGOPS European Workshop, System Support for Composing Distributed Applications. *Operating Systems Review*, 33(1):6–17, January 1999.

[24] F. Bao, R.H. Deng, Y. Han, A. Jeng, A.D. Narasimhalu, and T. Ngair. Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science, Security Protocols, Proceedings of 5th International Workshop*, pages 115—123, Paris, France, April 1997.

[25] A. Barnes, A. Hollway, and P.G. Neumann. Survivable computer-communication systems: The problem and working group recommendations. VAL-CE-TR-92-22 (revision 1). Technical report, U.S. Army Research Laboratory, AMSRL-SL-E, White Sands Missile Range, NM 88002-5513, May 1993. For Official Use Only.

[26] R.S. Barton. A critical review of the state of the programming art. In *Proceedings of the Spring Joint Computer Conference*, volume 23, pages 169–177, Montvale, New Jersey, May 1963. AFIPS Press.

[27] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison-Wesley, Reading, Massachusetts, 1998.

[28] D.E. Bell and L.J. La Padula. Secure computer systems : A mathematical model. Technical Report MTR-2547 Vol. II, Mitre Corporation, Bedford, Massachusetts, May 1973.

[29] D.E. Bell and L.J. La Padula. Secure computer systems : A refinement of the mathematical model. Technical Report MTR-2547 Vol. III, Mitre Corporation, Bedford, Massachusetts, December 1973.

[30] D.E. Bell and L.J. La Padula. Secure computer systems : Mathematical foundations. Technical Report MTR-2547 Vol. I, Mitre Corporation, Bedford, Massachusetts, March 1973.

[31] D.E. Bell and L.J. La Padula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The Mitre Corporation, Bedford, Massachusetts, May 1973.

[32] D.E. Bell and L.J. La Padula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, The Mitre Corporation, Bedford Massachusetts, March 1976.

[33] S.M. Bellovin. *Verifiably Correct Code Generation Using Predicate Transformers.* PhD thesis, Department of Computer Science, University of North Carolina at Chapel Hill, December 1982. (`http://www.research.att.com/~smb/dissabstract.html`).

[34] S.M. Bellovin. Probable plaintext cryptanalysis of the IP protocols. In *Proceedings of the Symposium on Network and Distributed System Security*, pages 52–59. Internet Society, February 1997.

[35] S.M. Bellovin and M. Merritt. Limitations of the Kerberos authentication system. In *USENIX Conference Proceedings, Winter '91*, January 1991. A version of this paper appeared in *Computer Communications Review,* October 1990.

[36] L.A. Benzinger, G.W. Dinolt, and M.G. Yatabe. Final report: A distributed system multiple security policy model. Technical report, Loral Western Development Laboratories, report WDL-TR00777, San Jose, California, October 1994.

[37] S. Berkovits, J.D. Guttman, and V. Swarup. Authentication for mobile agents. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science 1419, Mobile Agents and Security*, pages 114–136, 1998.

[38] K.J. Biba. Integrity considerations for secure computer systems. Technical Report MTR 3153, The Mitre Corporation, Bedford, Massachusetts, June 1975. Also available from USAF Electronic Systems Division, Bedford, Massachusetts, as ESD-TR-76-372, April 1977.

[39] K.J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, Massachusetts, April 1977.

[40] K.P. Birman and T.A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[41] M. Blaum, J. Bruck, K. Rubin, and W. Lenth. A coding approach for detection of tampering in write-once optical disks. *IEEE Transactions on Computers*, C-47(1):120–125, January 1998.

[42] M. Blume. *Hierarchical Modularity and Intermodule Optimization.* PhD thesis, Computer Science Department, Princeton University, November 1997.

[43] A.D. Blumenstiel. Security assessment considerations for ADL ADP systems. Technical report, U.S. Department of Transportation/RSPA/Volpe Center, Cambridge, Massachusetts, 1985.

[44] A.D. Blumenstiel. FAA computer security candidate countermeasures for electronic penetration of ADP systems. Technical report, U.S. Department of Transportation/RSPA/Volpe Center, Cambridge, Massachusetts, 1986.

[45] A.D. Blumenstiel. Potential consequences of and countermeasures for advanced automation system electronic penetration. Technical report, U.S. Department of Transportation/RSPA/Volpe Center, Cambridge, Massachusetts, 1986.

[46] A.D. Blumenstiel. Guidelines for National Airspace System electronic security. Technical report, U.S. Department of Transportation/RSPA/Volpe Center, Cambridge, Massachusetts, 1987.

[47] A.D. Blumenstiel. Federal Aviation Administration computer security plans. Technical report, U.S. Department of Transportation/RSPA/Volpe Center, produced by Science Resources Associates, Cambridge, Massachusetts, 1988.

[48] A.D. Blumenstiel. National Airspace System electronic security. Technical report, U.S. Department of Transportation/RSPA/Volpe Center, Cambridge, Massachusetts, 1988.

[49] A.D. Blumenstiel. Federal Aviation Administration AIS security accreditation guidelines. Technical report, National Institute on Standards and Technology, Gaithersburg, Maryland, 1990.

[50] A.D. Blumenstiel. Federal Aviation Administration AIS security accreditation application design. Technical report, U.S. Department of Transportation/RSPA/Volpe Center, Cambridge, Massachusetts, 1991.

[51] A.D. Blumenstiel. Federal Aviation Administration AIS security accreditation program instructions. Technical report, U.S. Department of Transportation/RSPA/Volpe Center, Cambridge, Massachusetts, 1992.

[52] A.D. Blumenstiel. Federal Aviation Administration sensitive application security accreditation guideline. Technical report, U.S. Department of Transportation/RSPA/Volpe Center, Cambridge, Massachusetts, 1992.

[53] A.D. Blumenstiel. Briefing on electronic security in the Communications, Navigation and Surveillance (CNS) environment. Technical report, U.S. Department of Transportation/RSPA/Volpe Center, Cambridge, Massachusetts, 1994.

[54] A.D. Blumenstiel and J. Itz. National Airspace System Data Interchange Network electronic security. Technical report, U.S. Department of Transportation/RSPA/Volpe Center, Cambridge, Massachusetts, 1988.

[55] A.D. Blumenstiel and P.E. Manning. Advanced Automation System vulnerabilities to electronic attack. Technical report, U.S. Department of Transportation/RSPA/Volpe Center, Cambridge, Massachusetts, July 1986.

[56] A.D. Blumenstiel et al. Federal Aviation Administration report to Congress on air traffic control data and communications vulnerabilities and security. Technical report, U.S. Department of Transportation/RSPA/Volpe Center, Cambridge, Massachusetts, 1993.

[57] A. Boswell. Specification and validation of a security policy model. *IEEE Transactions on Software Engineering*, 21(2):63–69, February 1995. Special section on Formal Methods Europe '93.

[58] K.A. Bradley, B. Mukherjee, R.A. Olsson, and N. Puketza. Detecting disruptive routers: A distributed network monitoring approach. In *Proceedings of the 1998 Symposium on Security and Privacy*, Oakland, California, May 1998. IEEE Computer Society.

[59] J. Brentano, S.R. Snapp, G.V. Dias, T.L. Goan, L.T. Heberlein, C.H. Ho, K.N. Levitt, and B. Mukherjee. An architecture for a distributed intrusion detection system. In *Fourteenth Department of Energy Computer Security Group Conference*, pages 25–45 in section 17, Concord, California, May 1991. Department of Energy.

[60] F.P. Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, Reading, Massachusetts, Second edition, 1995.

[61] E. Brynjolfsson and L.M. Hitt. Beyond the productivity paradox. *Communications of the ACM*, 41(8):11–12, August 1998.

[62] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[63] R.W. Butler. A survey of provably correct fault-tolerant clock synchronization techniques. Technical Report TM-100553, NASA Langley Research Center, February 1988.

[64] Canadian Systems Security Centre, Communications Security Establishment, Government of Canada. *Canadian Trusted Computer Product Evaluation Criteria, Version 3.0e*, January 1993.

[65] D.J. Carney. Quotations from Chairman David: A Little Red Book of truths to enlighten and guide on the long march toward the COTS revolution. Technical report, Carnegie-Mellon University Software Engineering Institute, Pittsburgh, Pennsylvania, 1998. `http://www.sei.cmu.edu/publications/documents/99.reports/lrb/little-red-book.html`

[66] D.M. Chess. Security issues in mobile code systems. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science 1419, Mobile Agents and Security*, pages 1–14, 1998.

[67] D.D. Clark et al. *Computers at Risk: Safe Computing in the Information Age.* National Research Council, National Academy Press, 2101 Constitution Ave., Washington, D.C. 20418, 5 December 1990. Final report of the System Security Study Committee.

[68] W.J. Clinton. The Clinton Administration's Policy on Critical Infrastructure Protection: Presidential Decision Directive 63. Technical report, U.S. Government White Paper, 22 May 1998.

[69] F. Cohen. Computer viruses. In *Seventh DoD/NBS Computer Security Initiative Conference*, pages 240–263. National Bureau of Standards, Gaithersburg, Maryland, 24–26 September 1984. Reprinted in Rein Turn (ed.), *Advances in Computer System Security*, Vol. 3, Artech House, Dedham, Massachusetts, 1988.

[70] F.J. Corbató. On building systems that will fail (1990 Turing Award Lecture, with a following interview by Karen Frenkel). *Communications of the ACM*, 34(9):72–90, September 1991.

[71] F.J. Corbató, J. Saltzer, and C.T. Clingen. Multics – the first seven years. In *Proceedings of the Spring Joint Computer Conference*, volume 40, Montvale, New Jersey, 1972. AFIPS Press.

[72] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, October 1971.

[73] C. Cowan and C. Pu. Survivability from a sow's ear: The retrofit security requirement. In *Position Papers for the 1998 Information Survivability Workshop — ISW '98*, pages 43–47, Orlando, Florida, 28–30 October 1998. IEEE.

[74] D. Craigen and K. Summerskill (eds.). *Formal Methods for Trustworthy Computer Systems (FM'89), A Workshop on the Assessment of Formal Methods for Trustworthy Computer Systems.* Springer-Verlag, Berlin, 1990. 23-27 July 1989, Nova Scotia, Canada.

[75] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[76] M. Crosbie and E.H. Spafford. Active defense of a computer system using autonomous agents. Technical report, Department of Computer Sciences, CSD-TR-95-008, Purdue University, West Lafayette IN, 1995.

[77] R.C. Daley and J.B. Dennis. Virtual memory, processes, and sharing in Multics. *Communications of the ACM*, 11(5), May 1968.

[78] R.C. Daley and P.G. Neumann. A general-purpose file system for secondary storage. In *AFIPS Conference Proceedings, Fall Joint Computer Conference*, pages 213–229. Spartan Books, November 1965.

[79] K.W. Dam and H.S. Lin, editors. *Cryptography's Role In Securing the Information Society.* National Research Council, National Academy Press, 2101 Constitution Ave., Washington, D.C. 20418, 1996. Final report of the Cryptographic Policy Study Committee, ISBN 0-309-05475-3.

[80] J.A. Davidson. Asymmetric isolation. In *Proceedings of the Twelfth Annual Computer Security Applications Conference*, pages 44–54, San Diego, California, December 1996. IEEE Computer Society.

[81] F. De Paoli, A.L. Dos Santos, and R.A. Kemmerer. Web browsers and security. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science 1419, Mobile Agents and Security*, pages 235–256, 1998.

[82] R.D. Dean. *Formal Aspects of Mobile Code Security.*
PhD thesis, Computer Science Department, Princeton University, January 1999.
(`http://www.cs.princeton.edu/~ddean/`)

[83] D.E. Denning, S.G. Akl, M. Heckman, T.F. Lunt, M. Morgenstern, P.G. Neumann, and R.R. Schell. Views for multilevel database security. *IEEE Transactions on Software Engineering*, 13(2), February 1987.

[84] Department of the Army. Joint Technical Architecture, version 5.0. Technical report, Office of the Secretary of the Army, September 1997.

[85] Y. Desmedt, Y. Frankel, and M. Yung. Multi-receiver/multi-sender network security: Efficient authenticated multicast/feedback. In *Proceedings of IEEE INFOCOM*. IEEE, 1992.

[86] W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(5), November 1976.

[87] E.W. Dijkstra. Co-operating sequential processes. In *Programming Languages, F. Genuys (editor)*, pages 43–112. Academic Press, 1968.

[88] E.W. Dijkstra. The structure of the THE multiprogramming system. *Communications of the ACM*, 11(5), May 1968.

[89] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[90] J.E. Dobson and B. Randell. Building reliable secure computing systems out of unreliable unsecure components. In *Proceedings of the 1986 Symposium on Security and Privacy*, pages 187–193, Oakland, California, April 1986. IEEE Computer Society.

[91] European Communities Commission. *Information Technology Security Evaluation Criteria (ITSEC), Provisional Harmonised Criteria (of France, Germany, the Netherlands, and the United Kingdom)*, June 1991. Version 1.2. Available from the Office for Official Publications of the European Communities, L-2985 Luxembourg, item CD-71-91-502-EN-C. Also available from UK CLEF, CESG Room 2/0805, Fiddlers Green Lane, Cheltenham UK GLOS GL52 5AJ, or GSA/GISA, Am Nippenkreuz 19, D 5300 Bonn 2, Germany.

[92] R.S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–412, July 1974.

[93] M. Feather. Rapid application of lightweight formal methods for consistency analyses. *IEEE Transactions on Software Engineering*, 24(11):949–959, November 1998.

[94] R.J. Feiertag and P.G. Neumann. The foundations of a provably secure operating system (PSOS). In *Proceedings of the National Computer Conference*, pages 329–334. AFIPS Press, 1979.

[95] E.W. Felten, D. Balfanz, D. Dean, and D.S. Wallach. Web spoofing: An Internet con game. In *Proceedings of the Nineteenth National Computer Security Conference*, pages 95–103, Baltimore, Maryland, 6–10 October 1997. NIST/NCSC.

[96] C. Fetzer and F. Cristian. Building fault-tolerant hardware clocks from COTS components. In *Proceedings of the 1999 Conference on Dependable Computing for Critical Applications*, pages 59–78, San Jose, California, January 1998.

[97] E.A. Feustel and T. Mayfield. The DGSA: Unmet information security challenges for operating system designers. *Operating Systems Review*, 32(1):3–22, January 1998.

[98] S. Forrest, editor. *Emergent Computation*, MIT Press, Cambridge, Massachusetts, 1991. Proceedings of the Ninth Annual CNLS Conference.

[99] Electronic Frontier Foundation. *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design.* O'Reilly and Associates, Sebastopol, California, 1998. See also the Risks Forum, volume 19, number 87, 17 July 1998.

[100] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[101] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The Digital distributed system security architecture. In *Proceedings of the Twelfth National Computer Security Conference*, pages 305–319, Baltimore, Maryland, 10–13 October 1989. NIST/NCSC.

[102] V.D. Gligor, S.I. Gavrila, and D. Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In *Proceedings of the 1998 Symposium on Security and Privacy*, Oakland, California, May 1998. IEEE Computer Society.

[103] A. Goldberg. A specification of Java loading and bytecode verification. In *Fifth ACM Conference on Computer and Communications Security*, pages 49–58, San Francisco, California, November 1998. ACM SIGSAC.

[104] L. Gong. A secure identity-based capability system. In *Proceedings of the 1989 Symposium on Research in Security and Privacy*, pages 56–63, Oakland, California, May 1989. IEEE Computer Society.

[105] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[106] L. Gong, R. Needham, and R. Yahalom. Reasoning about belief in cryptographic protocols. In *Proceedings of the 1990 Symposium on Research in Security and Privacy*, pages 234–248, Oakland, California, May 1990. IEEE Computer Society.

[107] L. Gong and R. Schemers. Implementing protection domains in the Java Development Kit 1.2. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, San Diego, California, March 1998.

[108] L. Gong and R. Schemers. Signing, sealing, and guarding Java objects. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science 1419, Mobile Agents and Security*, pages 206–216, 1998.

[109] R.S. Gray, D. Kotz, G. Cybenko, and D. Rus. D'agents: Security in a multiple-language, mobile-agent system. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science 1419, Mobile Agents and Security*, pages 154–187, 1998.

[110] P. Green. The art of creating reliable software-based systems using off-the-shelf software components. In *Proceedings of the Sixteenth International Symposium on Reliable Distributed Systems*, pages 118–120, Durham, North Carolina, 22-24 October 1997. IEEE Computer Society.

[111] I. Greenberg, P. Boucher, R. Clark, E.D. Jensen, T.F. Lunt, P.G. Neumann, and D. Wells. The multilevel secure real-time distributed operating system study. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, June 1992. Issued as Rome Laboratory report RL-TR-93-101, Rome Laboratory C3AB, Griffiss AFB NY 13441-5700. Contact Emilie Siarkiewicz, Internet: SiarkiewiczE@CS.RL.AF.MIL, phone 315-330-3241. For Official Use Only.

[112] JASA Standards Working Group. Joint airborne SIGINT architecture. Technical report, TASC, 131 National Business Parkway, Annapolis Junction, MD 20701, June-July 1998. Draft, Version 3; contact Paul L. Washington, Jr., 1-301-483-6000, ext. 2017, `PLWashington@jswg.org`.

[113] S. Halevi and H. Krawczyk. Public-key cryptography and password protocols. In *Fifth ACM Conference on Computer and Communications Security*, pages 122–131, San Francisco, California, November 1998. ACM SIGSAC.

[114] R.W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29:147–60, 1950.

[115] J.R. Heath, P.J. Kuekes, and R.S. Williams. A defect tolerant architecture for chemically assembled computers: The lessons of Teramac for the aspiring nanotechnologist. `http://neon.chem.ucla.edu/~schung/Hgrp/teramac.html`, 1997.

[116] T.L Heberlein, B. Mukherjee, and K.N. Levitt. A method to detect intrusive activity in a networked environment. In *Proceedings of the Fourteenth National Computer Security Conference*, pages 362–371, Washington, D.C., 1–4 October 1991. NIST/NCSC.

[117] C. Heitmeyer, J. Kirby, Jr., B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11):927–948, November 1998.

[118] H.M. Hinton. *Composable Safety and Progress Properties.* PhD thesis, University of Toronto, 1995.

[119] H.M. Hinton. Under-specification, composition, and emergent properties. In *Proceedings of the 1997 New Security Paradigms Workshop*, pages 83–93, Langdale, Cumbria, United Kingdom, September 1997. ACM SIGSAC.

[120] H.M. Hinton. Composing partially-specified systems. In *Proceedings of the 1998 Symposium on Security and Privacy*, Oakland, California, May 1998. IEEE Computer Society.

[121] F. Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science 1419, Mobile Agents and Security*, pages 92–113, 1998.

[122] C.M. Holloway, editor. *Third NASA Langley Formal Methods Workshop*, Hampton, Virginia, May 10-12 1995. NASA Langley Research Center. NASA Conference Publication 10176, June 1995.

[123] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[124] J. Horning and B. Randell. Process structuring. *ACM Computing Surveys*, 5(1), March 1973.

[125] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith, and B. Randell. A program structure for error detection and recovery. In *Operating Systems, Proceedings of an International Symposium, Notes in Computer Science 16*, pages 171–187. Springer-Verlag, Berlin, 1974.

[126] D.A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the IRE*, 40, 1952.

[127] D.A. Huffman. Canonical forms for information-lossless finite-state machines. *IRE Transactions on Circuit Theory (special supplement) and IRE Transactions on Information Theory (special supplement)*, CT-6 and IT-5:41–59, May 1959. A slightly revised version appeared in E.F. Moore, Ed., *Sequential Machines: Selected Papers*, Addison-Wesley, Reading, Massachusetts.

[128] IEEE. Standard specifications for public key cryptography. Technical report, IEEE Standards Department, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, Draft, June 1998. `http://grouper.ieee.org/groups/1363/`.

[129] R. Jagannathan and C. Dodd. GLU programmer's guide v0.9. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, November 1994. CSL Technical Report CSL-94-06.

[130] R. Jagannathan, T.F. Lunt, D. Anderson, C. Dodd, F. Gilham, C. Jalali, H.S. Javitz, P.G. Neumann, A. Tamaru, and A. Valdes. System Design Document: Next-generation Intrusion-Detection Expert System (NIDES). Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, 9 March 1993.

[131] G. Jakobson and M.D. Weissman. Alarm correlation. *IEEE Network*, pages 52–59, November 1993.

[132] H.S. Javitz and A. Valdes. The NIDES statistical component description and justification. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, March 1994.

[133] E.D. Jensen, J.A. Test, R.D. Reynolds, E. Burke, and J.G. Hanko. Alpha release 2 design summary report. Technical report 88120, Kendall Square Research Corporation, Cambridge, Massachusetts, September 1988.

[134] D.R. Johnson, F.F. Saydjari, and J.P. Van Tassel. MISSI security policy: A formal approach. Technical report, NSA R2SPO-TR001-95, 18 August 1995.

[135] M.F. Kaashoek and A.S. Tanenbaum. Fault tolerance using group communication. *ACM SIGOPS Operating System Review*, 25(2):71–74, April 1991.

[136] R. Kailar and V.D. Gligor. On the evolution of beliefs in authentication protocols. In *Proceedings of the IEEE Computer Security Foundations Workshop IV*, Franconia, New Hampshire, June 1991.

[137] R.Y. Kain and C.E. Landwehr. On access checking in capability-based systems. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, April 1986.

[138] M.H. Kang, J.N. Froscher, and I.S. Moskowitz. An architecture for multilevel secure interoperability. In *Proceedings of the Thirteenth Annual Computer Security Applications Conference*, pages 194–204, San Diego, California, December 1997. IEEE Computer Society.

[139] M.H. Kang, I. Moskowitz, B. Montrose, and J. Parsonese. A case study of two NRL pump prototypes. In *Proceedings of the Twelfth Annual Computer Security Applications Conference*, pages 32–43, San Diego, California, December 1996. IEEE Computer Society.

[140] P.A. Karger. Limiting the damage potential of discretionary Trojan horses. In *Proceedings of the 1987 Symposium on Security and Privacy*, pages 32–37, Oakland, CA, April 1987. IEEE Computer Society.

[141] P.A. Karger. *Improving Security and Performance for Capability Systems.* PhD thesis, Computer Laboratory, University of Cambridge, Cambridge, England, October 1988. Technical Report No. 149.

[142] G. Karjoth, D.B. Lange, and M. Oshima. A security model for aglets. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science 1419, Mobile Agents and Security*, pages 188–205, 1998.

[143] J. Kelsey, B. Schneier, and D. Wagner. Protocol interactions and the chosen protocol attacks. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science, Security Protocols, Proceedings of 5th International Workshop*, pages 91–104, Paris, France, April 1997.

[144] R.A. Kemmerer. Analyzing encryption protocols using formal verification techniques. *IEEE Journal on Selected Areas in Communications*, 7(4):448–457, May 1989.

[145] T. Kinberg. Debate: This house believes the development of robust distributed systems from components to be impossible. *Operating Systems Review*, 33(1):15–17, January 1999. The description of this debate appeared in "Report on the Eighth ACM SIGOPS European Workshop," System Support for Composing Distributed Applications, *loc.cit.*, pp. 6–17.

[146] S. Kliger, S. Yemini, Y. Yemini, D. Ohsie, and S. Stolfo. A coding approach to event correlation. In *Proceedings of the Fourth International Symposium on Integrated Network Management (IFIP/IEEE), Santa Barbara, CA, May 1995*, pages 266–277. Chapman & Hall, London, England, 1995.

[147] C. Ko. *Execution Monitoring of Security-Critical Programs in a Distributed System: A Specification-Based Approach.* PhD thesis, Computer Science Department, University of California at Davis, 1996.

[148] P.C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science, Advances in Cryptology, Proceedings of Crypto '96*, pages 104–113, Santa Barbara, California, August 1996.

[149] S.S. Kulkarni and A. Arora. Compositional design of multitolerant repetitive Byzantine agreement. *Proceedings of the Seventeenth International Conference on Foundations of Software Technology and Theoretical Computer Science, Kharagpur, India*, pages 169–183, December 1997.

[150] M.D. Ladue. When Java was one: Threats from hostile byte code. In *Proceedings of the Nineteenth National Computer Security Conference*, pages 104–115, Baltimore, Maryland, 6–10 October 1997. NIST/NCSC.

[151] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[152] L. Lamport. A simple approach to specifying concurrent program systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[153] L. Lamport, W.H. Kautz, P.G. Neumann, R.L. Schwartz, and P.M. Melliar-Smith. Formal techniques for fault tolerance in distributed data processing. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, April 1981. For Rome Air Development Center.

[154] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM TOPLAS*, 4(3):382–401, July 1982.

[155] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Operating Systems Review*, 25(5):165–182, October 1991. Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles.

[156] B.W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[157] B.W. Lampson. Redundancy and robustness in memory protection. In *Information Processing 74 (Proceedings of the IFIP Congress 1974)*, volume Hardware II, pages 128–132. North-Holland, Amsterdam, 1974.

[158] C.E. Landwehr, A.R. Bull, J.P. McDermott, and W.S. Choi. A taxonomy of computer program security flaws, with examples. Technical report, Center for Secure Information Technology, Information Technology Division, Naval Research Laboratory, Washington, D.C., November 1993.

[159] J.-C. Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *Digest of Papers, FTCS 15*, pages 2–11, Ann Arbor, Michigan, June 1985. IEEE Computer Society.

[160] G. Le Lann. Predictability in critical systems. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science, Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lyngby, Denmark, September 1998.

[161] G. Le Lann. Proof-based system engineering and embedded systems. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science, Embedded Systems*, 1998.

[162] W. Legato. Formal methods: Changing directions. In *Proceedings of the Eighteenth National Computer Security Conference*, Baltimore, Maryland, 10–13 October 1995. NIST/NCSC.

[163] N.G. Leveson. *Safeware: System Safety and Computers.* Addison-Wesley, Reading, Massachusetts, 1995.

[164] N.G. Leveson and M.P.E. Heimdahl. New approaches to critical-system survivability. In *Position Papers for the 1998 Information Survivability Workshop — ISW '98*, pages 111–114, Orlando, Florida, 28–30 October 1998. IEEE.

[165] K.N. Levitt, S. Crocker, and D. Craigen, editors. VERkshop III: Verification workshop. *ACM SIGSOFT Software Engineering Notes*, 10(4):1–136, August 1985.

[166] P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *Fifth ACM Conference on Computer and Communications Security*, pages 112–121, San Francisco, California, November 1998. ACM SIGSAC.

[167] P.D. Lincoln, N. Marti-Oliet, and J. Meseguer. Specification, transformation, and programming of concurrent systems in rewriting logic. Technical Report SRI-CSL-94-11, Computer Science Laboratory, SRI International, Menlo Park, California, May 1994.

[168] P.D. Lincoln and J.M. Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault Tolerant Computing Symposium 23*, pages 402–411, 1993.

[169] P.D. Lincoln and J.M. Rushby. Formally verified algorithms for diagnosis of manifest, symmetric, link, and Byzantine faults. Technical Report SRI-CSL-95-14, Computer Science Laboratory, SRI International, Menlo Park, California, October 1995.

[170] P.D. Lincoln, J.M. Rushby, N. Suri, and C. Walter. Hybrid fault algorithms. In *Proceedings of the Third NASA Langley Formal Methods Workshop, May 10-12, 1995*, pages 193–209. NASA Langley Research Center, June 1995.

[171] U. Lindqvist and P.A. Porras. Detecting computer and network misuse through the Production-Based Expert System Toolset (P-BEST). In *Proceedings of the 1999 Symposium on Security and Privacy*, Oakland, California, May 1999. IEEE Computer Society.

[172] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol: A comparison of two approaches. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science 1055, 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166, 1998.

[173] T. Lowman and D. Mosier. Applying the DoD Goal Security Architecture as a methodology for the development of system and enterprise security architectures. In *Proceedings of the Thirteenth Annual Computer Security Applications Conference*, pages 183–193, San Diego, California, December 1997. IEEE Computer Society.

[174] M. Lubaszewski and B. Courtois. A reliable fail-safe system. *IEEE Transactions on Computers*, C-47(2):236–241, February 1998.

[175] T.F. Lunt. Aggregation and inference: Facts and fallacies. In *Proceedings of the 1989 IEEE Symposium on Research in Security and Privacy*, May 1989.

[176] T.F. Lunt, R.R. Schell, W.R. Shockley, M. Heckman, and D. Warren. A near-term design for the SeaView multilevel database system. In *Proceedings of the 1988 Symposium on Security and Privacy*, pages 234–244, Oakland, California, April 1988. IEEE Computer Society.

[177] T.F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, C. Jalali, P.G. Neumann, H.S. Javitz, and A. Valdes. A Real-Time Intrusion-Detection Expert System (IDES). Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, 28 February 1992.

[178] T.F. Lunt and R.A. Whitehurst. The SeaView formal top level specifications and proofs. Final report, Computer Science Laboratory, SRI International, Menlo Park, California, January/February 1989. Volumes 3A and 3B of "Secure Distributed Data Views," SRI Project 1143.

[179] D. Malkhi, M.K. Reiter, and A.D. Rubin. Secure execution of Java applets using a remote playground. In *Proceedings of the 1998 Symposium on Security and Privacy*, Oakland, California, May 1998. IEEE Computer Society.

[180] A.P. Maneki. Algebraic properties of system composition in the Loral, Ulysses and McLean trace models. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, Kenmare, County Kerry, Ireland, June 1995.

[181] M. Mansouri-Samani and M. Sloman. Monitoring distributed systems. *IEEE Network*, pages 20–30, November 1993.

[182] T. Marsh (ed.). Critical Foundations: Protecting America's Infrastructures. Technical report, President's Commission on Critical Infrastructure Protection, October 1997.

[183] D. McCullough. Specifications for multi-level security and a hook-up property. In *Proceedings of the 1987 Symposium on Security and Privacy*, pages 161–166, Oakland, California, April 1987. IEEE Computer Society.

[184] D. McCullough. Noninterference and composability of security properties. In *Proceedings of the 1988 Symposium on Security and Privacy*, pages 177–186, Oakland, California, April 1988. IEEE Computer Society.

[185] D. McCullough. Ulysses security properties modeling environment: The theory of security. Technical report, Odyssey Research Associates, Ithaca, NY, July 1988.

[186] D. McCullough. A hookup theorem for multilevel security. *IEEE Transactions on Software Engineering*, 16(6), June 1990.

[187] G. McGraw and E.W. Felten. *Java Security: Hostile Applets, Holes, and Antidotes.* John Wiley and Sons, New York, 1997.

[188] G. McGraw and E.W. Felten. *Securing Java: Getting Down to Business with Mobile Code.* John Wiley and Sons, New York, 1997. This is the second edition of [187].

[189] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of the 1994 Symposium on Research in Security and Privacy*, pages 79–93, Oakland, California, May 1994. IEEE Computer Society.

[190] C. Meadows. A system for the specification and analysis of key management protocols. In *Proceedings of the 1991 Symposium on Research in Security and Privacy*, Oakland, California, May 1991. IEEE Computer Society.

[191] P.M. Melliar-Smith and L.E. Moser. Surviving network partitioning. *Computer*, 31(3):62–68, March 1998.

[192] P.M. Melliar-Smith and R.L. Schwartz. Formal specification and verification of SIFT: A fault-tolerant flight control system. *IEEE Transactions on Computers*, C-31(7):616–630, July 1982.

[193] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In *Research Directions on Concurrent Object-Oriented Programming.* MIT Press, Cambridge, Massachusetts, 1993.

[194] K. Meyer, M. Erlinger, J. Betser, C. Sunshine, G. Goldszmidt, and Y. Yemini. Decentralizing control and intelligence in network management. In *Proceedings of the Fourth International Symposium on Integrated Network Management (IFIP/IEEE), Santa Barbara, California, May 1995*, pages 4–16. Chapman & Hall, London, England, 1995.

[195] S. Micali. Fair public-key cryptosystems. In *Advances in Cryptology: Proceedings of CRYPTO '92 (E.F. Brickell, editor)*, pages 512–517. Springer-Verlag, Berlin, LCNS 740, 1992.

[196] J.K. Millen. Hookup security for synchronous machines. In *Proceedings of the IEEE Computer Security Foundations Workshop VII*, pages 2–10, Franconia, New Hampshire, June 1994. IEEE Computer Society.

[197] J.K. Millen. Local reconfiguration policies. In *Proceedings of the 1999 Symposium on Security and Privacy*, Oakland, California, May 1999. IEEE Computer Society. `http://www.csl.sri.com/~millen/reconfig.ps`".

[198] J.K. Millen. Survivability measure. Technical report, SRI International Computer Science Laboratory, Menlo Park, California, January 1999. `http://www.csl.sri.com/~millen/private/measure.ps`".

[199] S. Miller, B. Neuman, J. Schiller, and J. Saltzer. Kerberos authentication and authorization system. Technical report, MIT Project Athena Technical Plan Section E.2.1, 21 December 1987.

[200] J.C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murφ. In *Proceedings of the 1997 Symposium on Security and Privacy*, pages 141–151, Oakland, California, May 1997. IEEE Computer Society.

[201] E.F. Moore and C.E. Shannon. Reliable circuits using less reliable relays. *Journal of the Franklin Institute*, 262:191–208, 281–297, September, October 1956.

[202] R. Morris and K. Thompson. Password security: A case history. *Communications of the ACM*, 22(11):594–597, November 1979.

[203] R.T. Morris. Computer science technical report 117. Technical report, AT&T Bell Laboratories, Murray Hill, New Jersey, 25 February 1985.

[204] L. Moser, P.M. Melliar-Smith, and R. Schwartz. Design verification of SIFT. Contractor Report 4097, NASA Langley Research Center, Hampton, VA, September 1987.

[205] Mudge et al. Testimony of l0pht heavy industries. In *Weak Computer Security in Government: Is the Public at Risk? Hearing, Senate Hearing 105-609, ISBN 0-16-057456-0*, pages 71–91, Washington, D.C., 19 May 1998. U.S. Government Printing Office. Oral testimony is on pages 22–41.

[206] NASA Langley Research Center. *Formal Methods Specification and Verification, Volume I.* NASA, June 1995.

[207] NASA Langley Research Center. *Formal Methods Specification and Verification, Volume II.* NASA, Fall 1995.

[208] NCSC. *Trusted Network Interpretation Environments Guideline.* National Computer Security Center, 1 August 1990. NCSC-TG-011 Version-1.

[209] NCSC. *Trusted Network Interpretation (TNI).* National Computer Security Center, 31 July 1987. NCSC-TG-005 Version-1, Red Book.

[210] NCSC. *Trusted Database Management System Interpretation of the Trusted Computer System Evaluation Criteria (TDI).* National Computer Security Center, April 1991. NCSC-TG-021, Version-2, Lavender Book.

[211] NCSC. *Department of Defense Trusted Computer System Evaluation Criteria (TCSEC).* National Computer Security Center, December 1985. DOD-5200.28-STD, Orange Book.

[212] NCSC. *Guidance for Applying the Trusted Computer System Evaluation Criteria in Specific Environments.* National Computer Security Center, June 1985. CSC-STD-003-85, Yellow Book.

[213] G.C. Necula. *Compiling with Proofs.* PhD thesis, Computer Science Department, Carnegie-Mellon University, 1998.

[214] G.C. Necula and P. Lee. Research on proof-carrying code for untrusted-code security. In *Proceedings of the 1997 Symposium on Security and Privacy*, page 204, Oakland, California, May 1997. IEEE Computer Society.

[215] G.C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science 1419, Mobile Agents and Security*, pages 61–91, 1998.

[216] R.M. Needham and M.D. Schroeder. Using encryption for authentication and authorization systems. *Communications of the ACM*, 21(12):993–999, December 1978.

[217] B.C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9):33–38, September 1994.

[218] P.G. Neumann. Efficient error-limiting variable-length codes. *IRE Transactions on Information Theory*, IT-8:292–304, July 1962.

[219] P.G. Neumann. On a class of efficient error-limiting variable-length codes. *IRE Transactions on Information Theory*, IT-8:S260–266, September 1962.

[220] P.G. Neumann. Error-limiting coding using information-lossless sequential machines. *IEEE Transactions on Information Theory*, IT-10:108–115, April 1964.

[221] P.G. Neumann. The role of motherhood in the pop art of system programming. In *Proceedings of the ACM Second Symposium on Operating Systems Principles, Princeton New Jersey*, pages 13–18. ACM, October 1969.

[222] P.G. Neumann. System design for computer networks. In *Computer-Communication Networks (Chapter 2)*, pages 29–81. Prentice-Hall, 1971. N. Abramson and F.F. Kuo (eds.).

[223] P.G. Neumann. Computer security evaluation. In *AFIPS Conference Proceedings, NCC*, pages 1087–1095. AFIPS Press, January 1978. Reprinted in Rein Turn (ed.), *Advances in Computer Security*, Artech House, Dedham, Massachusetts, 1981.

[224] P.G. Neumann. On hierarchical design of computer systems for critical applications. *IEEE Transactions on Software Engineering*, SE-12(9), September 1986. Reprinted in Rein Turn (ed.), *Advances in Computer System Security*, Vol. 3, Artech House, Dedham, Massachusetts, 1988.

[225] P.G. Neumann. On the design of dependable computer systems for critical applications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, October 1990. CSL Technical Report CSL-90-10.

[226] P.G. Neumann. Rainbows and arrows: How the security criteria address computer misuse. In *Proceedings of the Thirteenth National Computer Security Conference*, pages 414–422, Washington, D.C., 1-4 October 1990. NIST/NCSC.

[227] P.G. Neumann. Can systems be trustworthy with software-implemented crypto? Technical report, Final Report, Project 6402, SRI International, Menlo Park, California, Menlo Park, California, October 1994. For Official Use Only, NOFORN.

[228] P.G. Neumann. *Computer-Related Risks.* ACM Press, New York, and Addison-Wesley, Reading, Massachusetts, 1994. ISBN 0-201-55805-X.

[229] P.G. Neumann. Architectures and formal representations for secure systems. Technical report, Final Report, Project 6401, SRI International, Menlo Park, California, Menlo Park, California, October 1995. CSL report 96-05.

[230] P.G. Neumann. Security risks in the emerging infrastructure. *Security in Cyberspace, Hearings, S. Hrg. 104-701*, pages 350–363, June 1996. ISBN 0-16-053913-7. Written testimony for the U.S. Senate Permanent Subcommittee on Investigations of the Senate Committee on Governmental Affairs. Oral testimony is on pages 106-111.

[231] P.G. Neumann. Computer security in aviation: Vulnerabilities, threats, and risks. In *International Conference on Aviation Safety and Security in the 21st Century, White House Commission on Safety and Security, and George Washington University*, January 13-15 1997. (http://www.csl.sri.com/neumann/air.html).

[232] P.G. Neumann. Computer-related infrastructure risks for federal agencies. In *Weak Computer Security in Government: Is the Public at Risk? Hearing, Senate Hearing 105-609, ISBN 0-16-057456-0*, pages 52–70, Washington, D.C., 19 May 1998. U.S. Government Printing Office. Oral testimony is on pages 5–22.

[233] P.G. Neumann. Illustrative risks to the public in the use of computer systems and related technology, index to RISKS cases. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, 1999. The most recent version is available on-line at `ftp://ftp.csl.sri.com/pub/users/neumann/illustrative.ps` and `.pdf` .).

[234] P.G. Neumann and P. Boucher. An evaluation of the security aspects of the Army Technical Architecture (ATA) document drafts. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, 2 January 1996. Final report, SRI Project 7104-200, for U.S. Army CECOM, Fort Monmouth, New Jersey.

[235] P.G. Neumann, R.S. Boyer, R.J. Feiertag, K.N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical report, Computer Science Laboratory SRI International, Menlo Park, California, May 1980. 2nd ed., Report CSL-116.

[236] P.G. Neumann, J. Goldberg, K.N. Levitt, and J.H. Wensley. A study of fault-tolerant computing. Final report for ARPA, AD 766 974, Stanford Research Institute, Menlo Park, CA, July 1973.

[237] P.G. Neumann and L. Lamport. Highly dependable distributed systems. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, June 1983. Final Report, Contract No. DAEA18-81-G-0062, for U.S. Army CECOM.

[238] P.G. Neumann and D.B. Parker. *A Study of Computer Abuse – Volume Two: Information Exploitation Database.* SRI International, Menlo Park, California, 31 July 1989. Final report for SRI Project 6812, U.S. Government.

[239] P.G. Neumann and D.B. Parker. A summary of computer misuse techniques. In *Proceedings of the Twelfth National Computer Security Conference*, pages 396–407, Baltimore, Maryland, 10–13 October 1989. NIST/NCSC.

[240] P.G. Neumann and D.B. Parker. *A Study of Computer Abuse – Volume One: Computer Abuse Techniques.* SRI International, Menlo Park, California, Revised, 2 March 1990. Final report for SRI Project 6812, U.S. Government.

[241] P.G. Neumann, N.E. Proctor, and T.F. Lunt. Preventing security misuse in distributed systems. Technical report, Computer Science Laboratory, SRI International, Menlo

Park, California, June 1992. Issued as Rome Laboratory report RL-TR-92-152, Rome Laboratory C3AB, Griffiss AFB NY 13441-5700. Contact Emilie Siarkiewicz, Internet: SiarkiewiczE@CS.RL.AF.MIL, phone 315-330-3241. For Official Use Only.

[242] P.G. Neumann, editor. VERkshop I: Verification Workshop. *ACM SIGSOFT Software Engineering Notes*, 5(3):4–47, July 1980.

[243] P.G. Neumann, editor. VERkshop II: Verification Workshop. *ACM SIGSOFT Software Engineering Notes*, 6(3):1–63, July 1981.

[244] W.L. O'Hern, Jr., task force chairman. An open systems process for DoD. Technical report, Open Systems Task Force, Defense Science Board, 1998? This report is still in draft form.

[245] E.I. Organick. *The Multics System: An Examination of Its Structure.* MIT Press, Cambridge, Massachusetts, 1972.

[246] J.K. Ousterhout, J.Y. Levy, and B.B. Welch. The Safe-Tcl security model. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science 1419, Mobile Agents and Security*, pages 217–234, 1998.

[247] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995. Special section on Formal Methods Europe '93.

[248] D.B. Parker. *Fighting Computer Crime.* John Wiley & Sons, New York, 1998.

[249] T.A. Parker. A secure European system for applications in a multi-vendor environment (The SESAME Project). In *Proceedings of the Fourteenth National Computer Security Conference*, pages 505–513, Washington, D.C., 1–4 October 1991. NIST/NCSC.

[250] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), December 1972.

[251] D.L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5), May 1972.

[252] D.L. Parnas. On a "buzzword": Hierarchical structure. In *Information Processing 74 (Proceedings of the IFIP Congress 1974)*, volume Software, pages 336–339. North-Holland, Amsterdam, 1974.

[253] D.L. Parnas. The influence of software structure on reliability. In *Proceedings of the International Conference on Reliable Software*, pages 358–362, April 1975. Reprinted with improvements in R. Yeh, *Current Trends in Programming Methodology I*, Prentice-Hall, 1977, 111–119.

[254] D.L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[255] D.L. Parnas. Mathematical descriptions and specification of software. In *Proc. of IFIP World Congress 1994, Volume I*, pages 354–359. IFIP, August 1994.

[256] D.L. Parnas. Software engineering: An unconsummated marriage. *Communications of the ACM*, 40(9):128, September 1997. *Inside Risks* column.

[257] D.L. Parnas, P.C. Clements, and D.M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–266, March 1985.

[258] D.L. Parnas and G. Handzel. More on specification techniques for software modules. Technical report, Fachbereich Informatik, Technische Hochschule Darmstadt, Research Report BS I 75/1, Germany, April 1975.

[259] D.L. Parnas, J. Madey, and M. Iglewski. Precise documentation of well-structured programs. *IEEE Transactions on Software Engineering*, 20(12):948–976, December 1994.

[260] D.L. Parnas and W.R. Price. The design of the virtual memory aspects of a virtual machine. In *Proceedings of the ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems.* ACM, March 1973.

[261] D.L. Parnas and D.L. Siewiorek. Use of the concept of transparency in the design of hierarchically structured systems. *Communications of the ACM*, 18(7):401–408, July 1975.

[262] J. Paul. Bugs in the program. Technical report, Report by the Subcommittee on Investigations and Oversight of the Committee on Science, Space and Technology, U.S. House of Representatives, 1990.

[263] L. Paulson. Mechanized proofs for a recursive authentication protocol. In *10th IEEE Computer Security Foundations Workshop*, pages 84–95. IEEE Computer Society, 1997.

[264] L. Paulson. Proving properties of security protocols by induction. In *10th IEEE Computer Security Foundations Workshop*, pages 70–83. IEEE Computer Society, 1997.

[265] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[266] R. Perlman. *Network Layer Protocols with Byzantine Robustness.* PhD thesis, MIT, Cambridge, Massachusetts, 1988.

[267] H. Petersen and M. Michels. On signature schemes with threshold verification detecting malicious verifiers. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science, Security Protocols, Proceedings of 5th International Workshop*, pages 67–77, Paris, France, April 1997.

[268] H. Petroski. *To Engineer is Human: The Role of Failure in Successful Design.* St. Martin's Press, New York, 1985.

[269] H. Petroski. *Design Paradigms: Case Histories of Error and Judgment in Engineering.* Cambridge University Press, Cambridge, England, 1994.

[270] H. Pfeifer, D. Schwier, and F.W. von Henke. Formal verification for time-triggered clock synchronization. In *Proceedings of the 1999 Conference on Dependable Computing for Critical Applications*, pages 193–212, San Jose, California, January 1998.

[271] C.P. Pfleeger. *Security in Computing.* Prentice-Hall, Englewood Cliffs, New Jersey, 1996. Second edition.

[272] S.L. Pfleeger. *Software Engineering: Theory and Practice.* Prentice-Hall, Englewood Cliffs, New Jersey, 1998.

[273] M. Pollitt. Cyberterrorism: Fact or fancy? In *Proceedings of the Nineteenth National Computer Security Conference*, pages 285–289, Baltimore, Maryland, 6–10 October 1997. NIST/NCSC.

[274] P.A. Porras. STAT: A State Transition Analysis Tool for intrusion detection. Master's thesis, Computer Science Department, University of California, Santa Barbara, July 1992.

[275] P.A. Porras and P.G. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *Proceedings of the Nineteenth National Computer Security Conference*, pages 353–365, Baltimore, Maryland, 22-25 October 1997. NIST/NCSC.

[276] P.A. Porras and A. Valdes. Live traffic analysis of TCP/IP gateways. In *Proceedings of the Symposium on Network and Distributed System Security.* Internet Society, March 1998.

[277] D. Prasad. *Dependable Systems Integration Using the Theories of Measurement and Decision Analysis.* PhD thesis, Department of Computer Science, University of York, August 1998.

[278] D. Prasad and J. McDermid. Dependability evaluation using a multi-criteria decision analysis procedure. In *To appear*, 1999.

[279] N.E. Proctor. SeaView formal specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, April 1991.

[280] N.E. Proctor and P.G. Neumann. Architectural implications of covert channels. In *Proceedings of the Fifteenth National Computer Security Conference*, pages 28–43, Baltimore, Maryland, 13–16 October 1992. (http://www.csl.sri.com/neumann/ncs92.html).

[281] B. Randell. System design and structuring. *Computer Journal*, 29(4):300–306, 1986.

[282] B. Randell and J.E. Dobson. Reliability and security issues in distributed computing systems. In *Proceedings of the Fifth Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles CA, January 1986.

[283] B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, editors. *Predictably Dependable Computing Systems.* Basic Research Series. Springer-Verlag, Berlin, 1995.

[284] T.R.N. Rao. *Error-Control Coding for Computer Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[285] M. Raynal. A case study of agreement problems in distributed systems: Non-blocking atomic commitment. In *Proceedings of the 1997 High-Assurance Systems Engineering Workshop*, pages 209–214, Washington, D.C., August 1997. IEEE Computer Society.

[286] D.D. Redell. *Naming and Protection in Extendible Operating Systems.* PhD thesis, University of California, 1974.

[287] D.D. Redell and R.S. Fabry. Selective revocation of capabilities. In *Proceedings of the International Workshop On Protection in Operating Systems*, pages 197–209, August 1974.

[288] M. Reiter and K. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems*, 16(3):986–1009, May 1994.

[289] J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. Technical report, University of Sussex, July 1998. ftp://ftp.cogs.susx.ac.uk/pub/reports/compsci/cs0498.ps.Z.

[290] J. Riordan and B. Schneier. Environmental key generation toward clueless agents. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science 1419, Mobile Agents and Security*, pages 15–24, 1998.

[291] R. Rivest and B. Lampson. SDSI – a simple distributed security infrastructure (version 1.0). Technical report, MIT Laboratory for Computer Science, September 1996. A later draft is available on-line (`http://theory.lcs.mit.edu/~cis/sdsi.html`).

[292] L. Robinson and K.N. Levitt. Proof techniques for hierarchically structured programs. *Communications of the ACM*, 20(4):271–283, April 1977.

[293] L. Robinson, K.N. Levitt, and B.A. Silverberg. *The HDM Handbook.* Computer Science Laboratory, SRI International, Menlo Park, California, June 1979. Three Volumes.

[294] J.A. Rochlis and M.W. Eichin. With microscope and tweezers: The Worm from MIT's perspective. *Communications of the ACM*, 32(6):689–698, June 1989.

[295] A.W. Roscoe and L. Wulf. Composing and decomposing systems under security properties. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, Kenmare, County Kerry, Ireland, June 1995.

[296] J.S. Rothfuss and J.W. Parrett. Go ahead, visit those Websites, you can't get hurt ... can you? In *Proceedings of the Nineteenth National Computer Security Conference*, pages 80–94, Baltimore, Maryland, 6–10 October 1997. NIST/NCSC.

[297] J.M. Rushby. A trusted computing base for embedded systems. In *Proceedings of the Seventh DoD/NBS Computer Security Initiative Conference*, pages 294–311, Gaithersburg, Maryland, September 1984.

[298] J.M. Rushby. Networks are systems. In *Proceedings of the Department of Defense Computer Security Center Invitational Workshop on Network Security*, pages 7–24 to 7–37, New Orleans, LA, March 1985. publ. by Department of Defense Computer Security Center.

[299] J.M. Rushby. Composing trustworthy systems. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, July 1991.

[300] J.M. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report SRI-CSL-92-2, SRI Computer Science Laboratory, Menlo Park, California, December 1992.

[301] J.M. Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In *Proceedings of the Thirteenth Conference on Principles of Distributed Computing*, pages 304–313, Los Angeles, California, August 1994. ACM.

[302] J.M. Rushby and B. Randell. A distributed secure system. *Computer*, 16(7):55–67, July 1983.

[303] J.M. Rushby and F. von Henke. Formal verification of the interactive convergence clock synchronization algorithm using EHDM. Technical Report SRI-CSL-89-3, Computer Science Laboratory, SRI International, Menlo Park, California, February 1989. Also available as NASA Contractor Report 4239.

[304] C. Salter, O.S. Saydjari, B. Schneier, and J. Wallner. Toward a secure system engineering methodology. Technical report, Draft was at `http://www.hokie.bs1.prc.com/ipa/PREPUB~2.html`, Undated draft, March 1998.

[305] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[306] T. Sander and C.F. Tschudin. Protecting mobile agents against malicious hosts. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science 1419, Mobile Agents and Security*, pages 44–60, 1998.

[307] T. Sander and C.F. Tschudin. Towards mobile cryptography. In *Proceedings of the 1998 Symposium on Security and Privacy*, Oakland, California, May 1998. IEEE Computer Society.

[308] S. Saniford-Chen and L.T. Heberlein. Holding intruders accountable on the Internet. In *Proceedings of the 1995 Symposium on Security and Privacy*, Oakland, California, May 1995. IEEE Computer Society.

[309] D. Santel, C. Trautmann, and W. Liu. The integration of a formal safety analysis into the software engineering process: An example from the pacemaker industry. In *Proceedings of the Symposium on the Engineering of Computer-Based Medical Systems*, pages 152–154, Minneapolis, MN, June 1988. IEEE Computer Society.

[310] F.B. Schneider. Understanding protocols for Byzantine clock synchronization. Technical Report 87-859, Department of Computer Science, Cornell University, Ithaca, NY, August 1987.

[311] F.B. Schneider. *On Concurrent Programming.* Springer Verlag, New York, 1997.

[312] F.B. Schneider and M. Blumenthal, editor. *Trust in Cyberspace.* National Research Council, National Academy Press, 2101 Constitution Ave., Washington, D.C. 20418, 1998. Final report of the National Research Council Committee on Information Trustworthiness.

[313] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C: Second Edition.* John Wiley and Sons, New York, 1996.

[314] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *Proceedings of the Seventh USENIX Security Symposium*, pages 53–62. USENIX, January 1998.

[315] B. Schneier and Mudge. Cryptanalysis of Microsoft's Point-to-Point Tunneling Protocol (PPTP). In *Fifth ACM Conference on Computer and Communications Security*, pages 132–141, San Francisco, California, November 1998. ACM SIGSAC.

[316] M.D. Schroeder. Cooperation of mutually suspicious subsystems in a computer utility. Technical report, Ph.D. Thesis, M.I.T., Cambridge, Massachusetts, September 1972.

[317] M.D. Schroeder, D.D. Clark, and J.H. Saltzer. The Multics kernel design project. In *Proceedings of the Sixth Symposium on Operating System Principles*, November 1977. ACM Operating Systems Review 11(5).

[318] M.D. Schroeder and J.H. Saltzer. A hardware architecture for implementing protection risks. *Communications of the ACM*, 15(3), March 1972.

[319] D. Seeley. Password cracking: A game of wits. *Communications of the ACM*, 32(6):700–703, June 1989.

[320] J.F. Shoch and J.A. Hupp. The "Worm" programs – early experience with a distributed computation. *Communications of the ACM*, 25(3):172–180, March 1982. Reprinted in Denning (ed.), *Computers Under Attack.*

[321] S.K. Shrivastava and F. Panzieri. The design of a reliable remote procedure call mechanism. *IEEE Transactions on Computers*, C-31(7):692–687, July 1982.

[322] O. Sibert, P.A. Porras, and R. Lindell. An analysis of the Intel 80x86 security architecture and implementations. *IEEE Transactions on Software Engineering*, SE-22(4), May 1996.

[323] D. Sidhu, A. Chung, and T.P. Blumer. Experience with formal methods in protocol development. *ACM SIGCOMM Computer Communication Review*, 21(2):81–101, April 1991.

[324] E.H. Spafford. The Internet Worm: crisis and aftermath. *Communications of the ACM*, 32(6):678–687, June 1989.

[325] SRI-CSL. EHDM *Specification and Verification System Version 4.1: Preliminary Definition of the* EHDM *Specification Language.* Computer Science Laboratory, SRI International, Menlo Park, California, 6 September 1988.

[326] M. Srivas and A. Camilleri, editors. *Formal Methods in Computer-Aided Design.* Springer-Verlag, Berlin, Lecture Notes in Computer Science, Vol. 1166, 1996.

[327] J.G. Steiner, C. Neuman, and J.I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter Conference*, pages 191–202, February 1988.

[328] D.E. Stevenson. Validation and verification methodologies for large scale simulations: There are no silver hammers, either. *To appear*, 1998.

[329] C. Stoll. Stalking the Wily Hacker. *Communications of the ACM*, 31(5):484–497, May 1988.

[330] C. Stoll. *The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage.* Doubleday, New York, 1989.

[331] D.W.J. Stringer-Calvert. *Mechanical Verification of Compiler Correctness.* PhD thesis, Department of Computer Science, University of York, 1998.

[332] K. Sullivan, J.C. Knight, X. Du, and S. Geist. Information survivability control systems. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE)*, 1999.

[333] D.I. Sutherland. A model of information flow. In *Proceedings of the Ninth National Computer Security Conference*, pages 175–183, September 1986.

[334] R. Thomas and R. Feiertag. Addressing survivability in the composable replaceable security services infrastructre. In *Position Papers for the 1998 Information Survivability Workshop — ISW '98*, pages 159–162, Orlando, Florida, 28–30 October 1998. IEEE.

[335] K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, August 1984.

[336] J.E. Tidswell and J.M. Potter. A dynamically typed access control model. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science 1438, Information Security and Privacy, Third Australasian Conference, ACISP'98, Brisbane, Australia*, pages 308–319, June 1998.

[337] J.T. Trostle. Timing attacks against trusted path. In *Proceedings of the 1998 Symposium on Security and Privacy*, Oakland, California, May 1998. IEEE Computer Society.

[338] UK-MoD. *Interim Defence Standard 00-55, The Procurement of Safety-Critical Software in Defence Equipment.* U.K. Ministry of Defence, 5 April 1991. DefStan 00-55; Part 1, Issue 1: Requirements; Part 2, Issue 1: Guidance.

[339] UK-MoD. *Interim Defence Standard 00-56, Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment.* U.K. Ministry of Defence, 5 April 1991. DefStan 00-56.

[340] US-Senate. *Security in Cyberspace.* U.S. Senate Permanent Subcommittee on Investigations of the Senate Committee on Governmental Affairs, Hearings, S. Hrg. 104-701, June 1996. ISBN 0-16-053913-7.

[341] N.H. Vaidya. A case for two-level recovery schemes. *IEEE Transactions on Computers*, 47(6):656–666, June 1998.

[342] G. Vigna. Cryptographic traces for mobile agents. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science 1419, Mobile Agents and Security*, pages 137–153, 1998.

[343] G. Vigna, ed. *Mobile Agents and Security.* Springer-Verlag, Berlin, Lecture Notes in Computer Science 1419, 1998.

[344] J.M. Voas and G. McGraw. *Software Fault Injection: Inoculating Programs Against Errors.* John Wiley and Sons, New York, 1998.

[345] D. Volpano and G. Smith. Language issues in mobile program security. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science 1419, Mobile Agents and Security*, pages 25–43, 1998.

[346] J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In *Automata Studies*, pages 43–98, Princeton University, Princeton, New Jersey, 1956.

[347] D.S. Wallach. *A New Approach to Mobile Code Security.* PhD thesis, Computer Science Department, Princeton University, January 1999. `http://www.cs.rice.edu/~dwallach/`.

[348] D.S. Wallach and E.W. Felten. Understanding Java stack inspection. In *Proceedings of the 1998 Symposium on Security and Privacy*, Oakland, California, May 1998. IEEE Computer Society.

[349] W.H. Ware. A retrospective of the criteria movement. In *Proceedings of the Eighteenth National Information Systems Security Conference*, pages 582–588, Baltimore, Maryland, 10-13 October 1995. NIST/NCSC.

[350] J.H. Wensley et al. Design study of software-implemented fault-tolerance (SIFT) computer. NASA contractor report 3011, Computer Science Laboratory, SRI International, Menlo Park, California, June 1982.

[351] I.S. Winkler. Position paper. In *Position Papers for the 1998 Information Survivability Workshop — ISW '98*, pages 189–191, Orlando, Florida, 28–30 October 1998. IEEE.

[352] W.D. Young and J. McHugh. Coding for a believable specification to implementation mapping. In *Proceedings of the 1987 Symposium on Security and Privacy*, pages 140–148, Oakland, CA, April 1987. IEEE Computer Society.

[353] J. Zabarsky. Failure recovery for distributed processes in single system image clusters. In *Springer-Verlag, Berlin, Lecture Notes in Computer Science 1388, Parallel and Distributed Processing*, pages 564–583, Berlin, Germany, April 1998.

[354] A. Zakinthinos and E.S. Lee. The composability of non-interference. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, Kenmare, County Kerry, Ireland, June 1995.

[355] A. Zakinthinos and E.S. Lee. Composing secure systems that have emergent properties. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, pages 117–122, Rockport, Massachusetts, June 1998.

Index

Abstraction, 7, 97
  horizontal, 97
  vertical, 97
ACM
  standards, 81
Alternative routing, 10
Analysis
  real-time, 80
  static, 80
Architecture
  abstract (DGSA), 94
  generic (DGSA), 94
  limited-trust, 102
  logical (DGSA), 94
  operational (JTA), 89
  structural, 96
  systems (JTA), 89
  technical (JTA), 89
Assurance, 5
Auditability, 52
Authentication, 51
  avoiding fixed passwords, 36
  flaws, 34
  in TCSEC, 85
  pervasive, 98
  servers, 106
  single sign-on, 36
  subsystems, 106
  systemic, 61
Availability, 4, 51
  multilevel, 13

Backup
  trustworthy, 61
BAN Logic, 75
Barnes, Anthony, 131
Barton, R.S.
  representation of structure, 72
Black Hawk
  interference, 24
Blumenstiel, Alexander D., 44
Bootloading
  integrity, 106
Brooks, Frederick P.
  Mythical Man Month, 127
  throw one away, 127
Byzantine
  authentication, 10
  faults, 10
  protocols, 10

Causes of risks, 26
Certificate
  authorities, 63
  public-key, 63
Chaining Layered Integrity Checks, 106
CIAO, 84
Clocks
  dependable, 11
Commitment
  nonblocking, 10
  two-phase, 10
Common Criteria, 85
Common Intrusion Detection Format (CIDF), 108
Composability, 98
Composition
  generalized, 8
  modular, 7, 71
Compromisibility, 14
  from below, 15
  from outside, 15
  from within, 15
Computers at Risk, 8, 84
Confidentiality, 51
Control systems, 103

COSPO Switched Workstation, 103
Covert channels, 38
  avoidance, 52, 139
  in trusted paths, 51
Criteria
  Common Criteria, 85
  CTCPEC, 85
  ITSEC, 85
  TCSEC, 85
Cryptography
  applications, 105
  cracking DES, 62
  fair public-key, 10
  hardware implemented, 105
  in software, 51
  in systems and networks, 8
  key management, 105
  protocols, 62
  public key
    RSA, 41, 105
  public-key, 105
    Diffie-Hellman, 105
  secret-key, 105
  shared-key, 105
  software implemented, 105
  threshold schemes, 10
CTCPEC, 85
CTSS security exposure, 16

DDSSA, 106
Dean, Drew, 131
  hashing weaknesses, 36
  JSM, 112
  PhD thesis, 110
Decomposition
  modular, 7, 71
Deep Crack, 62
Defenses
  against compromise, 19
Deming, W. Edwards, 122
Dependability, 5, 6
Dependence
  compromisibility of, 14
  generalized, 9
  strict, 9
    avoidance of, 99
Detection
  of compromise, 7
  of misuse, 52
    in real time, 80
DGSA, 91, 93
Diffie-Hellman, 105
Diffie-Hellman public-key cryptography, 105
Dijkstra, Edsger W.
  early work, 71
  THE system, 139
Distributed Crack, 63
Diversity
  in architecture, 67
  in design, 98
DNSSEC, 62
Domains, 11
  distributed, 7
DSS, 102

Education and training, 83
Effects of Risks, 26
Einstein, Albert
  education and experience, 138
  problem solving, 67
  rules of work, 1
  simplicity, 2
  wisdom, 138
Electromagnetic interference
  Black Hawk, 24
  House testimony, 20
  Patriot, 24
  Predator, 24
  Tomahawk case?, 23
EMERALD, 80, 107
  use of structure, 104
Emergent properties, 12
  human safety, 57
  in HDM, 12
  survivability, xi
  undesirable behavior and
    incomplete specification, 13
Encapsulation, 97
Encryption, 105
Endogirding, 19

Error-correcting codes, 9
Exogirding, 19

Fault injection
  cryptanalysis, 62
  software testing, 74
Fault tolerance, 10
  requirements, 52
  techniques, 79
Firewalls, 11, 101
Formal methods
  for authentication, 75
  for cryptographic protocols, 75
  for dependability, 6
  for design, 72
  for fault tolerance, 74
  for security, 74
  for specification, 72
  for survivability, 74
  for synchronization, 12

GASSP, 101
Generalized
  composition, 8
  dependence, 9
  multilevel survivability, 13
  survivability, 12
    requirements, 49
Girding against compromise, 19
GLU, 82
GNU System, 77
Guards, 11, 101

Hierarchical Development Methodology (HDM), 12, 139
  emergent properties, 12
  hierarchical composition, 73
Hierarchy
  absence of, 16
  of abstraction, 14, 97
  of locking protocols, 10
  of survivability requirements, 47
  PSOS layering, 9
Horning, James, 96
Human safety
  as an emergent property, 57

IEEE
  standards, 81
IETF
  standards, 81
Inheritance, 71
Integrity, 51
  checks for, 11
  multilevel, 13
  of bootloading, 106
  of resources, 106
Interoperability, 7
IP
  security architecture, 62
IP Version 6, 62, 69
IPSEC, 62, 69
Isolation, 101
ITSEC, 85

Java, 112
  Development Kit (JDK), 112
  Security Manager (JSM), 112
  Virtual Machine (JVM), 112
Joint Airborne SIGINT Architecture (JASA), 94
Joint Technical Architecture (JTA), 88
JSM, 112

Karger, Paul
  discretionary access controls, 112
Keller, Helen, 30
Kerberos, 77, 106
Kernels
  MLS, 101
  separation, 101
Key management, 105
Kocher, Paul
  differential power analysis on crypto, 62
  timing attacks on cryptographic implementations, 62

L0pht, 63
Lampson, Butler W., 139
  confinement, 112, 139
  SDSI, 106
Least common mechanism, 99

Least privilege, 98
Lincoln, Patrick A.
  Byzantine clocks, 11
Linux, 77
Loading
  dynamic, 108
  of resources, 106
Locking
  hierarchical, 10

Maude, 83
Meadows, Catherine, 74
MEII, 103
Millen, Jonathan, 131
  survivability research, 74
Misuse detection, 11
Mobile code, 100, 108
  dynamic linking and loading, 108
  security architecture, 110
  security controls, 110
  security risks, 37
Monitoring, 107
Morris, Bob
  dictionary attacks, 36
MSL, Multiple single-level, 102
Multics, 139
  domains, 112
  rings, 111
  use of structure, 104
Multilevel
  availability, 13
  integrity, 13, 102
  security, 13, 102
  survivability, 13, 102
Mutual suspicion, 11

National Research Council report
  Computers at Risk, 8
  Cryptography's Role in Securing the Information Society, 8
  Trust in Cyberspace, 8
National Research Laboratory
  MSL efforts, 102
Necula, George
  PhD thesis, 110
Needham-Schroeder, 106
Networking, 105
Newcastle
  Distributed Secure System (DSS), 102
NIPC, 84

Object-oriented paradigm, 71, 98
  abstraction, 71
  encapsulation, 71
  inheritance, 71
  polymorphism, 71
Open Group, 81
Open Source Program Office COSPO, 103
Open Systems Task Force (OSTF), 95
Open-source software, 75
  examples, 77
  motivation, 136
  risks, 77
  robust, 78
OSTF, 95

Parnas, David L.
  uses relation, 9
  early papers, 71
  engineering discipline, 71
  information hiding, 127
Passwords
  AR 380-19 policy, 92
  attacks, 35
  capture by Trojan horse, 38
  CTSS exposure, 16
  denial-of-service attack, 25
  dictionary attack, 36, 41
  JTA5.0 policy, 93
  one-time, 77
  preencryptive analysis, 38
  sniffing, 118
Patriot
  clock drift, 24
  EMI problems, 24
PCCIP
  follow-ons, 84
  infrastructure survivability, 8
  organizational entities, 83
  report, x, 3

  research recommendations, 128
PDD-62, 84
PDD-63, 84
Pentium Floating Divide flaw, 16
Performance, 5
  as a subrequirement, 53
  threats, 42
Perimeter
  of survivability, 104
  of trustworthiness, 13, 104
PGP, 77
Polymorphism, 71
Porras, Phillip, 131
Portable computing, 100
Predator
  EMI problems, 24
Prevention
  of compromise, 7
  of misuse, 51
Principles
  for security, 101
  least common mechanism, 99
  least privilege, 98
  object-oriented, 71
  of cryptographic protocols, 75
  separation, 99
  structural, 97
Procurement, 65
Proof-carrying code, 11, 68, 75, 107, 110
Proprietary code, 76
  problems, 65
Protocols, 105
  ARPAnet routing, 10
  Byzantine, 10
  cryptographic, 62, 68, 75
  DNSSEC, 62
  for networks, 61
  IP Version 6, 62
  IP vulnerabilities, 17
  need for survivability, 4, 69, 124
  unsecure interactions among, 72
  upgraded, 105
PSOS, 139
  hierarchical layering, 9
  object orientation, 71
  use of structure, 104
Public-key infrastructure (PKI), 63

Rainbow books: TCSEC, 85
Randell, Brian, 96
  Newcastle DSS, 102
Read-only memory, 102
Real-time
  accountability, 56
  availability, 56
Recovery Blocks, 10
Recovery blocks, 103
Reliability, 4
  as a subrequirement, 52
  out of unreliable components, 9
  threats, 42
Requirements
  for availability, 4
  for performance, 5
  for reliability, 4
  for security, 4
  for survivability, 46
  interdependencies, 46
Research and development, 81
Resource
  loading, 106
Reverse engineering
  protection against, 51
Revocation
  of mobile code, 110, 113
RFCs, 62
Risks
  involving survivability, 30
  sources, 19
Risks Forum, 26, 62, 84, 126
Rivest, Ron
  SDSI, 106
Rivest-Shamir-Adleman (RSA), 105
RSA, 41, 105
Run-time checks, 11
Rushby, John M.
  isolation kernel, 139
  Newcastle DSS, 102

Saltzer, J.H., 139

Scalability, 7
Schopenhauer, Arthur
  generality vs. experience, 125, 138
Schroeder, Michael D., 139
  PhD thesis, 112
  protection, 139
Scrutability, 100
SDSI, 106
SeaView, 139
  use of structure, 104
Security, 4
  as a subrequirement, 50
  attack modes, 38
  kernels, 11, 101
  multilevel, 13
  multilevel databases, 11
  threats, 30
Self-synchronization, 9
Separation
  kernels, 101
  of domains, 99
  of duties, 99
  of policy and mechanism, 98
  of roles, 99
Server
  files, 107
  names, 107
  network, 107
SESAME, 106
Silk-purse sow's-ear metaphor, 119
Simple Maude, 82
Software
  fault injection, 74
  open-source, 77
Software engineering, 70
SPKI, 106
Standards, 81
Starlight Interactive Link, 103
Structure
  architectural, 96
  organizing principles, 97
Survivability
  application, 50
  as an emergent property, xi, 57
  as an overall system property, 54
  case histories of outages, 21
  curricula, 132
  definition, 2
  dependencies, 14
  enterprise, 50
  generalized, 12
    as an emergent property, 12
  information, 49
  multilevel, 13, 82, 102
  network, 50
  perimeter, 104
  requirements, 49
  risks, 30
  subrequirements, 47
  syllabus, 132
  system, 50
  threats, 30
Syllabus for survivability, 132
Synchronization
  robust, 10
System development
  failures, 64
  practice, 70

TAFIM, 93
Tamperproofing, 51
Tao Te Ching, 88
TCBs, 101
TCSEC, 85
  accountability, 113
  auditing, 113
  centralized view, 101
  incompleteness, 86
  interpretations, 92
  limitations, 85
Teramac, 66
Testing, 74
Thompson, Ken
  C compiler Trojan horse, 17, 36, 77
  dictionary attacks, 36
Threats
  to performance, 42
  to reliability, 42
  to security, 30
  to survivability, 30

- Tolerance
  - of compromise, 7
  - of faults, 4
  - of human foibles, 137
- Transactions
  - fulfillment, 10
- Trojan horses, 36
- Trust, 5
- Trust in Cyberspace, 8
- Trusted path
  - absence of, 61
  - for backup and retrieval, 106
  - for bootloading, 106
  - for reconfiguration, 106
  - realization, 106
  - requirements, 51
- Trustworthiness, 5
  - enhancement, 9
  - localized, 91
  - of backup, 61
  - perimeter, 13, 104
  - unknown, 11

- Undergirding, 19
- Unified Cryptologic Architecture (UCA), 95

- Vigna, Giovanni, 111
- Viruses, 41

- Walczak, Paul, 131
- Wallach, Dan S.
  - JSM, 112
  - PhD thesis, 110
- WORA, 100, 110
- WOVORA, 110
- Write once, run anywhere (WORA), 100
- Write-once, verify-once, run-anywhere (WOVORA), 110